Imperial College London

Department of Computing

GiMMiK - Generating Bespoke Matrix Multiplication Kernels for Various Hardware Accelerators; Applications in High-Order Computational Fluid Dynamics

Author:
Bartosz D. Wozniak

Supervisor:
Prof. Paul H. J. Kelly
Co-Supervisor:
Dr. Peter E. Vincent

June 17, 2014

Abstract

Matrix multiplication is a fundamental linear algebra routine ubiquitous in all areas of science and engineering. Highly optimised BLAS libraries (cuBLAS and clBLAS on GPUs) are the most popular choices for an implementation of the General Matrix Multiply (GEMM) in software. However, performance of library GEMM is poor for small matrix sizes. In this thesis we consider a *block-by-panel* type of matrix multiplication, where the block matrix is typically small (e.g. dimensions of 96×64), motivated by an application in PyFR– the most recent implementation of Flux Reconstruction schemes for high-order fluid flow simulations on unstructured meshes. We show how prior knowledge of the operator matrix can be exploited to generate highly performant kernel code, which outperforms the cuBLAS and clBLAS GEMM implementations. We present GiMMiK– a generator of bespoke matrix multiplication kernels for the CUDA and OpenCL platforms. GiMMiK generates code by fully unrolling the matrix-vector product. The generated kernels embed values of the operator matrix directly in the code to benefit from the use of the constant cache and compiler optimisations. Further, we reduce the number of floating-point operations by removing multiplications by zeros. We are able to achieve speedups for individual PyFR matrices of up to 9.98 (12.20) times on the Tesla K40c and 63.30 (13.07) times on the GTX 780 Ti in double (single) precision. Using GiMMiK as the matrix multiplication kernel provider allows us to achieve a speedup of up to 1.72 (2.14) for an example simulation of an unsteady flow over a cylinder executed with PyFR in double (single) precision on the Tesla K40c.

A general paper *"GiMMiK- Generating Bespoke Matrix Multiplication Kernels for Various Hardware Accelerators; Applications in High-Order Computational Fluid Dynamics"* by Bartosz D. Wozniak, Freddie D. Witherden, Peter E. Vincent and Paul H. J. Kelly has been prepared for submission to the *Computer Physics Communications* journal, based on the findings in this report. It is available upon request.

Acknowledgements

I would like to thank Prof. Paul Kelly, Dr. Peter Vincent and Freddie Witherden for their dedication and the amount of time they have contributed to this project. I am grateful to Dr. David Ham and Dr. Francis Russell for providing the initial guidance to stir the project in the right direction. Lastly, I would like to acknowledge and thank my friends and family for their unrelenting support and guidance throughout my time at Imperial.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Matrix multiplication is ubiquitous in all spheres of science and engineering, hence the need for efficient and performant implementations of such operations in software. A lot of effort has been put into building and optimising *Basic Linear Algebra Subprograms* (BLAS) libraries. *General Matrix Multiplication* (GEMM) subroutine of level-3 BLAS is among the most popular choices for an implementation of the matrix product. However, GEMM is very generic and usually performs best with large problem sizes [6, 11, 7, 12]. In situations where the matrices are known a priori, a faster implementations can be achieved. In this thesis we are interested in developing a highly performant matrix product routine for a *block-by-panel* (see Figure 1.1) type of matrix multiplication, where the operator matrix is typically small (e.g. 96×64 elements). This is motivated by an application in Flux Reconstruction [9] schemes for high-order fluid flow simulations on unstructured grids. However, we believe that the usefulness of our research goes beyond the area of Computational Fluid Dynamics (CFD) and can impact other fields of engineering as well.

In this thesis we present GiMMiK– a generator of matrix multiplication kernels. GiMMiK analyses a given operator matrix and generates optimised and highly performant CUDA and OpenCL kernel code that can run across a variety of hardware accelerators.

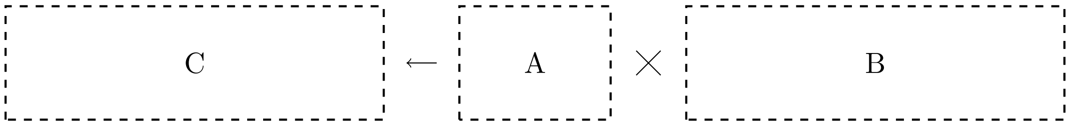


Figure 1.1: Diagram representing block-by-panel type of matrix multiplication. In this type of matrix product the operator matrix is typically small and square, while the operand and output matrices are fat.

1.1 Context

Within the area of Computational Fluid Dynamics (CFD) there is a growing need for efficient and accurate high-order schemes for numerical simulations. High-order methods can potentially deliver better accuracy at a similar computational cost to low-order methods. Unfortunately, existing high-order methods are less robust and harder to implement than their low-order counterparts, which prevents their adaptation in industry and to a lesser extent in academia.

In 2007 Huynh [9] presented the Flux Reconstruction (FR) approach, a unifying mathematical framework allowing an efficient development of high-order schemes. The details of this approach are described in Section 2.1. Huynh showed how well-known high-order schemes such as Discontinuous Galerkin (DG) methods and Spectral Difference (SD) methods can be cast within the Flux Reconstruction framework. In 2009 Huynh [10] showed how FR can be applied to diffusion problems. Most importantly, the FR framework allows for development of new schemes with favourable properties. Their main advantages over traditional high-order schemes are improved robustness, accuracy, stability and simplicity of implementation. Further work by Vincent et al. [20] and Castonguay et al. [3] resulted in a new class of energy-stable schemes for solving conservation laws problems for quadrilateral, hexahedral and triangular element meshes. Williams et al. [22] have extended the schemes to tetrahedral elements. Castonguay et al. [4] in 2011 were the first to present a high-order compressible viscous flow solver for mixed unstructured grids based on the Flux Reconstruction approach, designed to run on clusters of GPUs.

The most recent development in the area of Flux Reconstruction is PyFR, an open-source framework for solving advection-diffusion type problems on streaming architectures [23]. The very nature of Flux Reconstruction methods allows to cast many of the computation steps into matrix-matrix multiplication operations as described in Section 2.2. For this reason a GPU implementation of these schemes is very attractive, as the devices exhibit inherently high floating-point performance and memory bandwidth, suitable for the arithmetically intensive linear algebra operations. There is a number of highly optimised BLAS libraries, which can be employed to compute the required matrix products. NVIDIA cuBLAS GEMM [14] or the OpenCL clBLAS GEMM [1] are the obvious candidates for the GPU platform. However, already in 2011 Castonguay et al. [4] identified the need for bespoke matrix multiplication kernels to achieve high performance of their solver. The problem with available, highly optimised BLAS libraries is not their sub-optimal implementation but rather their general nature. Experimental data suggests that BLAS GEMM performs especially well and achieves near peak performance for large, square matrices [6, 7].

Each time step of the simulation performed by PyFR amounts to a repeated application of the same set of five operator matrices to the data, combined with some element-local transformations [23]. The exact characteristics of these matrices depend on many numerical method choices i.e. the shape and dimensionality of the mesh elements, the desired order of accuracy and the type of equations used to solve the problem. Casting the computation steps to matrix multiplication operations enables us to navigate all the numerical scheme choices freely, without incurring any performance penalty due to an unoptimised implementation of the solver.

The parameters of the numerical schemes dictate the size of the operator matrices, which is typically small. For hexahedral meshes it ranges from (4×8) to (96×64) and up to (1029×343) for the first, third and sixth order of accuracy correspondingly. The full specification of the characteristics of the matrices used by the PyFR solver across 1–6 orders of accuracy is available in Appendix A and further discussed in Section 2.3. The operator matrices stay constant for the duration of the simulation and are also known in advance, which opens up an opportunity to analyse them and generate bespoke, highly specialised kernels for each matrix to improve over the performance of state-of-the-art BLAS libraries.

1.2 Objectives

The aim of this project is to investigate the performance improvement achievable through the use of bespoke matrix multiplication kernels over the state-of-the-art BLAS GEMM implementations for a block-by-panel type of matrix multiplication characteristic to PyFR. We pick CUDA and OpenCL as the development platforms of choice and will evaluate the performance of our optimisations on two modern industry-grade GPUs: Tesla K40c form NVIDIA and FirePro W9100 from AMD and also on a consumer-grade NVIDIA GeForce GTX 780 Ti to further explore the applicability of our optimisations on commodity hardware.

In Chapter 4 we describe the methodology used to generate our bespoke multiplication kernels and the various software optimisations we have attempted. We have taken a systematic approach to evaluate each of the proposed optimisations in order to incorporate the successful ones into GiMMiK. The studied techniques involve loop unrolling, sparsity elimination, and common sub-expression elimination. We have also investigated the relative advantages and drawbacks of using different types of available memory to store the operator matrix. We aim to present a comprehensive set of evidence for the success or failure of the proposed optimisations obtained through benchmarking of our kernels on a well-diversified suite of matrices with different sizes and sparsity patterns.

Chapter 5 presents the empirical analysis of GiMMiK's kernels and gives the final assessment of the performance improvements our kernels are able to realise over cuBLAS and clBLAS GEMM for individual matrices. As the last step we demonstrate the usefulness of our proposed solution by plugging our kernels into PyFR and investigating the final performance improvement the solver is able to achieve during an example compressible fluid flow simulation on an unstructured mesh.

1.3 Contributions

The following list summarizes the contributions of this thesis:

- We show how, with a prior knowledge of the operator matrix, we are able to generate matrix multiplication kernel code, which performs better than state-of-the-art cuBLAS and clBLAS GEMM. We achieve speedups of up to 9.98 (12.20) times on the Tesla K40c and 63.30 (13.07) times on the GTX 780 Ti in double (single) precision for individual PyFR matrices in the block-by-panel type of product.

- We present GiMMiK– an open-source Python library for generating matrix multiplication kernels for CUDA and OpenCL platforms available for download at `https://github.com/bartwozniak/GiMMiK`.

- We propose a series of software optimisation techniques, which we speculate can bring performance improvements when applied to our CUDA and OpenCL kernels and incorporate the successful ones into GiMMiK. In a systematic way each optimisation is exhaustively evaluated on a well-diversified set of operator matrices extracted from the PyFR solver. The benchmark spans a range of matrix sizes and sparsity patterns.

- By incorporating GiMMiK into PyFR we are able to grant significant performance improvements of up to 1.72 (2.14) in double (single) precision on a single Tesla K40c, which allows us to reduce the computational time of an exemplary compressible unsteady flow simulation from a matter of weeks to a matter of days. Through this performance improvement we can further influence the numerical method choices and allow for better quality results.

A general paper *"GiMMiK - Generating Bespoke Matrix Multiplication Kernels for Various Hardware Accelerators; Applications in High-Order Computational Fluid Dynamics"* by Bartosz D. Wozniak, Freddie D. Witherden, Peter E. Vincent and Paul H. J. Kelly has been prepared for submission to the *Computer Physics Communications* journal, based on the findings in this report. It is available upon request. We believe that the methodology applied in this study can give a valuable insight into efficient implementations of small-scale linear algebra kernels on GPUs.

Chapter 2

Background

In this chapter we summarize the basic principles behind the Flux Reconstruction approach to high-order fluid flow simulations implemented by PyFR, which is the motivating subject of our investigation. In Section 2.2 we show how operations performed in the FR schemes can be cast into matrix multiplication problems. Later, in Section 2.3 we characterise the operator matrices used in PyFR and explain how they vary with the numerical method choices made for each simulation. At the end of this chapter, we give an introduction to General Purpose Computing on GPUs and explain the basics of CUDA and OpenCL programming models (Section 2.4). Lastly, we describe the three state-of-the-art BLAS libraries, which are predominantly used on the GPU platform and will serve as the benchmark for our investigation.

2.1 Flux Reconstruction

The following subsections will describe the basic steps underlying the Flux Reconstruction approach and demonstrate how it can be applied to one and multi dimensional domains as described by Castonguay et al. [4].

2.1.1 Flux Reconstruction Approach in 1D

For the purpose of 1D domains let us consider the following 1D scalar conservation law equation within an arbitrary domain $\mathbf{\Omega}$:

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = 0 \tag{2.1}$$

where x is a spatial coordinate, t is time, $u = u(x,t)$ is a conserved scalar quantity and $f = f(u, \frac{\partial u}{\partial x})$ is the flux in the x direction. Now consider partitioning $\mathbf{\Omega}$ into N

non-overlapping elements $\mathbf{\Omega}_n$

$$\mathbf{\Omega} = \bigcup_{n=1}^{N} \mathbf{\Omega}_n. \tag{2.2}$$

Within each element $\mathbf{\Omega}_n$ we will represent the exact solution u by a function $u_n^\delta = u_n^\delta(x,t)$ which is a degree p polynomial within the element and zero outside. Similarly, we will represent the exact flux f by a function $f_n^\delta = f_n^\delta(x,t)$ which is a degree $p+1$ polynomial within the element and zero outside. Thus, the total approximate solution u^δ and flux f^δ over the domain $\mathbf{\Omega}$ can be written as

$$u^\delta = \sum_{n=1}^{N} u_n^\delta \approx u, \quad f^\delta = \sum_{n=1}^{N} f_n^\delta \approx f. \tag{2.3}$$

To simplify the implementation, it is advantageous to cast each $\mathbf{\Omega}_n$ to a standard element $\mathbf{\Omega}_S = \{\xi | -1 \leq \xi \leq 1\}$ via an invertible mapping $\Theta_n(\xi)$.

$$x = \Theta_n(\xi) = \left(\frac{1-\xi}{2}\right) x_n + \left(\frac{1+\xi}{2}\right) x_{n+1} \tag{2.4}$$

This mapping allows us to solve the following transformed equation within the standard element

$$\frac{\partial \hat{u}^\delta}{\partial t} + \frac{\partial \hat{f}^\delta}{\partial \xi} = 0 \tag{2.5}$$

where

$$\hat{u}^\delta = \hat{u}^\delta(\xi,t) = J_n u_n^\delta(\Theta_n(\xi),t), \quad \hat{f}^\delta = \hat{f}^\delta(\xi,t) = f_n^\delta(\Theta_n(\xi),t)$$

and $J_n = (x_{n+1} - x_n)/2$.

The Flux Reconstruction approach can be applied to equation (2.5) and consists of seven stages. The first stage defines a set of $p+1$ solution points within the standard element $\mathbf{\Omega}_S$ and specifies the form of the approximate solution $\hat{u}^\delta$ as a polynomial of degree p of the form

$$\hat{u}^\delta = \sum_{i=1}^{p+1} \hat{u}_i^\delta l_i \tag{2.6}$$

where l_i is the 1D Lagrange polynomial associated with the i^{th} solution point and $\hat{u}_i^\delta$ represent the value of $\hat{u}^\delta$ at the solution point ξ_i.

In the second stage a common interface solution at the two ends of an element are calculated. To do this we calculate the approximate solution at the boundaries using equation (2.6). The common interface solution denoted $\hat{u}^{\delta I}$ can be computed using values form both sides of the interface. The exact methodology for calculating the interface solutions depends on the nature of the equations that are being solved.

The third stage involves the construction of a corrected solution gradient $\hat{q}^\delta$, which approximates the solution gradient within the element. In order to define $\hat{q}^\delta$ we start by considering correction functions $g_L(\xi)$ and $g_R(\xi)$, which in some sense approximate zero within $\mathbf{\Omega}_S$ and satisfy the following

$$g_L(-1) = 1, \quad g_L(1) = 0,$$

$$g_R(-1) = 0, \quad g_L(1) = 1,$$

$$g_L(\xi) = g_R(-\xi).$$

The corrected gradient $\hat{q}^\delta$ is defined as

$$\hat{q}^\delta = \frac{\partial \hat{u}^\delta}{\partial \xi} + (\hat{u}_L^{\delta I} - \hat{u}_L^\delta)\frac{\partial g_L}{\partial \xi} + (\hat{u}_R^{\delta I} - \hat{u}_R^\delta)\frac{\partial g_R}{\partial \xi} \tag{2.7}$$

where $\hat{u}_L^{\delta I}$ and $\hat{u}_R^{\delta I}$ are the transformed common solutions at the left and right interfaces obtained in the previous step and $\hat{u}_L^\delta = \hat{u}^\delta(-1)$ and $\hat{u}_R^\delta = \hat{u}^\delta(1)$ are the values of the approximate solution at the left and right interfaces obtained from equation (2.6). The exact form of g_L and g_R will not be considered in the thesis.

The fourth stage is concerned with the definition of the approximate transformed discontinuous flux within element $\mathbf{\Omega}_S$ denoted $\hat{f}^{\delta D}$. It is defined at the solution points described in the first stage and can be computed as

$$\hat{f}^{\delta D} = \sum_{i=1}^{p+1} \hat{f}_i^{\delta D} l_i \tag{2.8}$$

where the coefficient $\hat{f}_i^{\delta D}$ is the value of the transformed flux at the solution point ξ_i evaluated from the approximate solution $\hat{u}^\delta$ and the corrected gradient $\hat{q}^\delta$.

The fifth stage involves calculating the numerical interface fluxes at either end of the standard element $\mathbf{\Omega}_S$. To do so we must first obtain the approximate solution and the corrected gradient at these points within each element using equations (2.6) and (2.7). The exact way of computing the common interface fluxes using values from both sides of the interface again depends on the nature of the equations that are being solved.

In the sixth step, correction flux $\hat{f}^{\delta C}$ is added to the discontinuous flux $\hat{f}^{\delta D}$. We require the summation to be equal to the interface flux found in the previous stage. For this we define correction functions $h_L(\xi)$ and $h_R(\xi)$ analogous to those from stage three. Likewise, they need to satisfy

$$h_L(-1) = 1, \quad h_L(1) = 0,$$

$$h_R(-1) = 0, \quad h_L(1) = 1,$$

$$h_L(\xi) = h_R(-\xi).$$

The correction flux can now be defined as

$$\hat{f}^{\delta C} = (\hat{f}_L^{\delta I} - \hat{f}_L^{\delta D})h_L + (\hat{f}_R^{\delta I} - \hat{f}_R^{\delta D})h_R, \tag{2.9}$$

where $\hat{f}_L^{\delta D} = \hat{f}^{\delta D}(-1)$ and $\hat{f}_R^{\delta D} = \hat{f}^{\delta D}(1)$. The approximate total transformed flux $\hat{f}^{\delta}$ can be constructed as follows

$$\hat{f}^{\delta} = \hat{f}^{\delta D} + \hat{f}^{\delta C} = \hat{f}^{\delta D} + (\hat{f}_L^{\delta I} - \hat{f}_L^{\delta D})h_L + (\hat{f}_R^{\delta I} - \hat{f}_R^{\delta D})h_R. \tag{2.10}$$

The last stage involves calculating divergence of $\hat{f}^{\delta}$ at the solution points using the expression

$$\frac{\partial \hat{f}^{\delta}}{\partial \xi}(\xi_i) = \sum_{j=1}^{p+1} \hat{f}_j^{\delta D} \frac{\partial l_j}{\partial \xi}(\xi_i) + (\hat{f}_L^{\delta I} - \hat{f}_L^{\delta D})\frac{\partial h_L}{\partial \xi}(\xi_i) + (\hat{f}_R^{\delta I} - \hat{f}_R^{\delta D})\frac{\partial h_R}{\partial \xi}(\xi_i), \tag{2.11}$$

which can be used to approximate the evolution of $\hat{u}^{\delta}$ in time by using a suitable temporal discretionary of

$$\frac{\mathrm{d}\hat{u}_i^{\delta}}{\mathrm{d}t} = -\frac{\partial \hat{f}^{\delta}}{\partial \xi}(\xi_i). \tag{2.12}$$

2.1.2 Flux Reconstruction Approach in 2D

This section will describe how Flux Reconstruction can be applied to quadrilateral elements as it was first detailed by Vincent et al. [20]. The extension to hexahedral elements is straightforward. For the purpose of demonstrating the Flux Reconstruction approach to 2D quadrilateral elements let us consider the 2D scalar conservation law

$$\frac{\partial u}{\partial t} + \nabla_{xy} \cdot \boldsymbol{f} = 0, \tag{2.13}$$

with an arbitrary domain $\boldsymbol{\Omega}$, where $\boldsymbol{f} = (f, g)$, $f = f(u, \nabla u)$ is the flux in the x direction and $g = g(u, \nabla u)$ is the flux in the y direction. Again, we partition the domain into N non-overlapping quadrilateral elements $\boldsymbol{\Omega}_n$ such that

$$\boldsymbol{\Omega} = \bigcup_{n=1}^{N} \boldsymbol{\Omega}_n. \tag{2.14}$$

Similarly to the 1D case we will transform the element into a standard element with a mapping

$$\begin{pmatrix} x \\ y \end{pmatrix} = \sum_{i=1}^{K} M_i(\xi, \eta) \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \tag{2.15}$$

where K is the number of points used to define the shape of the physical element, (x_i, y_i) are the coordinates of the points and $M_i(\xi, \eta)$ are the shape functions. After this transformation the evolution of u_n^δ within $\boldsymbol{\Omega}_n$ can be solved using

$$\frac{\partial \hat{u}^\delta}{\partial t} + \nabla_{\xi\eta} \cdot \hat{\boldsymbol{f}}^\delta = 0, \tag{2.16}$$

where

$$\hat{u}^\delta = J_n u_n^\delta(\Theta_n(\xi, \eta), t), \tag{2.17}$$

$$\hat{\boldsymbol{f}}^\delta = (\hat{f}^\delta, \hat{g}^\delta) = \left(\frac{\partial y}{\partial \eta} f_n^\delta - \frac{\partial x}{\partial \eta} g_n^\delta, \frac{\partial y}{\partial \xi} f_n^\delta - \frac{\partial x}{\partial \xi} g_n^\delta\right) \tag{2.18}$$

and J_n, $\frac{\partial x}{\partial \xi}$, $\frac{\partial x}{\partial \eta}$, $\frac{\partial y}{\partial \xi}$ and $\frac{\partial y}{\partial \eta}$ depend on the shape of element n and can be evaluated using equation (2.15).

Next, for quadrilateral elements, $(p+1)^2$ solution points are defined within the standard element and $(p+1)$ flux points on each edge (total of $4(p+1)$). The approximate solution can be written as

$$\hat{u}^\delta = \sum_{\substack{i=1\\j=1}}^{p+1} \hat{u}_{i,j}^\delta l_i(\xi) l_j(\eta), \tag{2.19}$$

where $l_i(\xi)$ and $l_j(\eta)$ are the 1D Lagrange polynomials associated with the 1D solution point at (ξ_i, η_i).

The corrected gradient $\hat{\boldsymbol{q}}^\delta = (\hat{q}_\xi^\delta, \hat{q}_\eta^\delta)$ consists of a component in each direction ξ and η and is obtained using the 1D correction functions (g_L, g_R) and (g_B, g_T) as

$$\begin{aligned} \hat{q}_\xi^\delta(\xi_i, \eta_j) &= \frac{\partial \hat{u}^\delta}{\partial \xi}(\xi_i, \eta_j) + (\hat{u}_L^{\delta I} - \hat{u}_L^\delta)\frac{\partial g_L}{\partial \xi}(\xi_i) + (\hat{u}_R^{\delta I} - \hat{u}_R^\delta)\frac{\partial g_R}{\partial \xi}(\xi_i) \\ \hat{q}_\eta^\delta(\xi_i, \eta_j) &= \frac{\partial \hat{u}^\delta}{\partial \eta}(\xi_i, \eta_j) + (\hat{u}_B^{\delta I} - \hat{u}_B^\delta)\frac{\partial g_B}{\partial \eta}(\eta_j) + (\hat{u}_T^{\delta I} - \hat{u}_T^\delta)\frac{\partial g_T}{\partial \eta}(\eta_j), \end{aligned} \tag{2.20}$$

where $\hat{u}_R^{\delta I}$, $\hat{u}_L^{\delta I}$, $\hat{u}_T^{\delta I}$ and $\hat{u}_B^{\delta I}$ are the transformed common interface values of the approximate solution at the flux points located along the lines $\xi = \xi_i$ and $\eta = \eta_j$. The values of the solution at the flux points within each element ($\hat{u}_L^\delta$, $\hat{u}_R^\delta$, $\hat{u}_B^\delta$ and $\hat{u}_T^\delta$) are computed using equation (2.19). The corrected gradient in the entire element is then constructed as

$$\hat{\boldsymbol{q}}^\delta = \sum_{\substack{i=1\\j=1}}^{p+1} \hat{\boldsymbol{q}}_{i,j}^\delta l_i(\xi) l_j(\eta). \tag{2.21}$$

Values for discontinuous flux at the solution points ($\hat{\boldsymbol{f}}_{i,j}^{\delta D}$) can be found directly from the approximate solution $\hat{u}^\delta$ and the corrected gradient $\hat{\boldsymbol{q}}^\delta$ and hence we obtain the formula for discontinuous flux

$$\hat{\boldsymbol{f}}^{\delta D}(\xi,\eta) = \sum_{\substack{i=1\\j=1}}^{p+1} \hat{\boldsymbol{f}}^{\delta D}_{i,j} l_i(\xi) l_j(\eta). \tag{2.22}$$

The divergence of the discontinuous flux is thus

$$\begin{aligned}\nabla_{\xi\eta}\cdot\hat{\boldsymbol{f}}^{\delta D}(\xi,\eta) &= \frac{\partial \hat{f}^{\delta D}}{\partial \xi} + \frac{\partial \hat{g}^{\delta D}}{\partial \eta} \\ &= \sum_{\substack{i=1\\j=1}}^{p+1} \hat{\boldsymbol{f}}^{\delta D}_{i,j} \frac{\partial l_i(\xi)}{\partial \xi} l_j(\eta) + \sum_{\substack{i=1\\j=1}}^{p+1} \hat{\boldsymbol{f}}^{\delta D}_{i,j} l_i(\xi) \frac{\partial l_j(\eta)}{\partial \eta}.\end{aligned} \tag{2.23}$$

The divergence of the transformed correction flux $\nabla_{\xi\eta}\cdot\hat{\boldsymbol{f}}^{\delta C} = \frac{\partial \hat{f}^{\delta C}}{\partial \xi} + \frac{\partial \hat{g}^{\delta C}}{\partial \eta}$ at the solution point (ξ_i,η_j) is computed with the 1D methodology in each direction as

$$\begin{aligned}\frac{\partial \hat{f}^{\delta C}}{\partial \xi}(\xi_i,\eta_j) &= (\hat{f}^{\delta I}_L - \hat{f}^{\delta D}_L)\frac{\partial h_L}{\partial \xi}(\xi_i) + (\hat{f}^{\delta I}_R - \hat{f}^{\delta D}_R)\frac{\partial h_R}{\partial \xi}(\xi_i) \\ \frac{\partial \hat{g}^{\delta C}}{\partial \eta}(\xi_i,\eta_j) &= (\hat{g}^{\delta I}_B - \hat{g}^{\delta D}_B)\frac{\partial h_B}{\partial \eta}(\eta_j) + (\hat{g}^{\delta I}_T - \hat{g}^{\delta D}_T)\frac{\partial h_T}{\partial \eta}(\eta_j),\end{aligned} \tag{2.24}$$

where $\hat{f}^{\delta I}_L$, $\hat{f}^{\delta I}_R$, $\hat{g}^{\delta I}_B$ and $\hat{g}^{\delta I}_T$ are the transformed common interface fluxes computed at the flux points located along lines $\xi = \xi_i$ and $\eta = \eta_j$. The transformed discontinues fluxes at the flux points within each element ($\hat{f}^{\delta D}_L$, $\hat{f}^{\delta D}_R$, $\hat{g}^{\delta D}_B$ and $\hat{g}^{\delta D}_T$) are computed using equation (2.22).

Analogous to the 1D case, the total transformed flux is found as a sum of a discontinuous component $\hat{\boldsymbol{f}}^{\delta D}$ and a correction component $\hat{\boldsymbol{f}}^{\delta C}$,

$$\hat{\boldsymbol{f}}^{\delta} = \hat{\boldsymbol{f}}^{\delta D} + \hat{\boldsymbol{f}}^{\delta C}. \tag{2.25}$$

Using equations (2.25), (2.23) and (2.24) we can progress the solution in time with

$$\frac{\mathrm{d}\hat{u}^{\delta}_{i,j}}{\mathrm{d}t} = -\left(\frac{\partial \hat{f}^{\delta}}{\partial \xi}(\xi_i,\eta_j) + \frac{\partial \hat{g}^{\delta}}{\partial \eta}(\xi_i,\eta_j)\right). \tag{2.26}$$

For brevity, the case of triangular elements will not be discussed in this thesis, but can be found in [3].

2.2 Matrix Representation of FR Schemes

The aim of this section is to illustrate how Flux Reconstruction schemes can be cast to a form allowing for an efficient implementation on GPUs using matrix multiplications. For a more detailed explanation see [4, 23]. For this purpose let us use the compressible Navier-Stokes equations as an example. Consider the matrix $[\hat{U}_s]$ of dimensions $N_s \times 5$ (where N_s is the number of solution points per element and 5 is the number of Navier-Stokes equations), which stores the approximate solution at the solution points.

$$[\hat{U}_s] = \begin{bmatrix} \hat{\rho}_1^\delta & \hat{\rho u}_1^\delta & \hat{\rho v}_1^\delta & \hat{\rho w}_1^\delta & \hat{\rho e}_1^\delta \\ \hat{\rho}_2^\delta & \hat{\rho u}_2^\delta & \hat{\rho v}_2^\delta & \hat{\rho w}_2^\delta & \hat{\rho e}_2^\delta \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \hat{\rho}_{N_s}^\delta & \hat{\rho u}_{N_s}^\delta & \hat{\rho v}_{N_s}^\delta & \hat{\rho w}_{N_s}^\delta & \hat{\rho e}_{N_s}^\delta \end{bmatrix} \tag{2.27}$$

Further, consider the matrix $[\hat{U}_f]$ of dimensions $N_f \times 5$ (where N_f is the number of flux points per cell) that stores the approximate solution at the flux points.

$$[\hat{U}_f] = \begin{bmatrix} \hat{\rho}_1^\delta & \hat{\rho u}_1^\delta & \hat{\rho v}_1^\delta & \hat{\rho w}_1^\delta & \hat{\rho e}_1^\delta \\ \hat{\rho}_2^\delta & \hat{\rho u}_2^\delta & \hat{\rho v}_2^\delta & \hat{\rho w}_2^\delta & \hat{\rho e}_2^\delta \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \hat{\rho}_{N_f}^\delta & \hat{\rho u}_{N_f}^\delta & \hat{\rho v}_{N_f}^\delta & \hat{\rho w}_{N_f}^\delta & \hat{\rho e}_{N_f}^\delta \end{bmatrix} \tag{2.28}$$

The first step in the Flux Reconstruction schema is to compute the discontinuous approximate solution $\hat{u}^\delta$ at the flux points from the solution points. We can do this in the following operation

$$[\hat{U}_f^D] = M_0[\hat{U}_s], \tag{2.29}$$

where M_0 is of dimension $N_f \times N_s$ and depends on the type of elements used and is the same for all elements of the same type. To find the values of the common solution at the cell interfaces we need to loop over all the flux point pairs and compute the value $\hat{u}^{\delta I}$. Allow $[\hat{U}_f^I]$ to denote the transformed common interface solution for all elements.

Now, to compute the corrected gradient $\hat{\boldsymbol{q}}^\delta$ we yet need to find the discontinuous solution gradient $\hat{\nabla}\hat{u}^\delta$. Consider defining a matrix $[\hat{Q}_s^D]$ of dimension $(3N_s) \times 5$ to store the discontinuous gradient at the solution points, which can be obtained from

$$[\hat{Q}_s^D] = M_4[\hat{U}_s], \tag{2.30}$$

where M_4 is of dimension $(3N_s) \times N_s$ and again depends on the type of elements used and is the same for all elements of the same type. After the transformed common approximation value ($[\hat{U}_f^I]$) has been calculated for each flux point pair, the corrected gradient can be computed for the solution points using the following equation

$$[\hat{Q}_s] = M_6\left([\hat{U}_f^I] - [\hat{U}_f^D]\right) + [\hat{Q}_s^D], \tag{2.31}$$

where matrix M_6 is of dimension $3N_s \times N_f$.

Next, we want to compute the transformed discontinuous flux $\hat{\boldsymbol{f}}^{\delta D}$ at the solution points. This operation depends on the approximate solution $\hat{u}^\delta$ and the corrected gradient $\hat{\boldsymbol{q}}^\delta$, which were computed in the previous steps. It is computed independently for each point in the mesh and the results are stored in a matrix denoted by $[\hat{F}_s^D]$. The discontinuous flux is then used to compute the divergence using

$$[(div\hat{F})_s^D] = M_1[\hat{F}_s^D], \tag{2.32}$$

where M_1 is of dimension $N_s \times (3N_s)$.

In order to evaluate the common interface flux we require the discontinuous solution and the corrected gradient at the flux points within each element. The corrected gradient at the flux points can be computed with the M_0 matrix as follows

$$[\hat{Q}_f] = M_5[\hat{Q}_s] \qquad \text{where } M_5 = \begin{bmatrix} M_0 & 0 & 0 \\ 0 & M_0 & 0 \\ 0 & 0 & M_0 \end{bmatrix}. \tag{2.33}$$

To find the common values of the interface flux we need to loop over all the flux point pairs. The result is stored in matrix $[\hat{F}_f^I]$.

To compute the correction flux $\hat{\boldsymbol{f}}^{\delta C}$ we need to find the discontinuous flux at the flux points denoted by the matrix $[\hat{F}_f^D]$, which can be obtained from

$$[\hat{F}_f^D] = M_2[\hat{F}_s^D], \tag{2.34}$$

where M_2 is of dimension $N_f \times 3N_s$.

In the penultimate step, before the solution is progressed in time, the divergence of the total approximate flux is computed using the following formula

$$[(div\hat{F})_s] = M_3\left([\hat{F}_f^I] - [\hat{F}_f^D]\right) + [(div\hat{F})_s^D], \tag{2.35}$$

where M_3 is of dimension $N_f \times N_s$.

Witherden et al. [23] in their paper on PyFR show that these steps can be easily extended and applies to any number of dimensions and other element types. Further, they demonstrate how some of the matrix operations can be reordered and grouped together to achieve better performance in their implementation. Consider equation (2.31), which can be rewritten using equations (2.29) and (2.30) in the following way

$$[\hat{Q}_s] = M_6 \left([\hat{U}_f^I] - M_0[\hat{U}_s]\right) + M_4[\hat{U}_s], \tag{2.36}$$

and simplified to

$$[\hat{Q}_s] = M_6[\hat{U}_f^I] + (M_4 - M_6M_0)\,[\hat{U}_s]. \tag{2.37}$$

This results in a new matrix $M_{460} = M_4 - M_6M_0$, which can be computed before the simulation and reduces the number of arithmetic operations required to produce the desired solution. Similarly, equation (2.35) can be rewritten using equations (2.32) and (2.34) in the following way

$$[(div\hat{F})_s] = M_3 \left([\hat{F}_f^I] - M_2[\hat{F}_s^D]\right) + M_1[\hat{F}_s^D], \tag{2.38}$$

and simplified to

$$[(div\hat{F})_s] = M_3[\hat{F}_f^I] + (M_1 - M_3M_2)\,[\hat{F}_s^D]. \tag{2.39}$$

This again results in a new matrix $M_{132} = M_1 - M_3M_2$, which can be found prior to the computation and hence reduce the number of required arithmetic operations. Labelling of the operator matrices M_0 - M_6 is the naming convention used in PyFR and will be used throughout the rest of this thesis.

2.3 Characteristics of Operator Matrices in FR

Consider matrix multiplication of the form:

$$C \leftarrow \alpha AB + \beta C,$$

where A is the operator matrix, B is the data and C is the output. The operator matrices described in Section 2.2 all exhibit a similar set of characteristics. Their size depends on the exact equations that are being solved, the type of the elements they are evaluated on and the degree of accuracy (number of solution or flux points within each element). For quadrilateral and hexahedral elements the operator matrices are sparse due to the tensor product formulation on the solution points within each element, while for triangular and tetrahedral elements they are dense. Furthermore, these operator matrices are known a priori and remain constant for the duration of the computation. They are typically small and square, the size ranges from 6×3 to 1029×343 across 1–6 orders of accuracy. From a practical point of view, the most relevant matrices are those for the third order of accuracy with dimensions 96×64. The width of the B matrix depends on the number of elements in the mesh. In this investigation we have used B

of $50,000$ elements wide for all benchmarking runs. This is representative of a small, non-trivial fluid flow simulation. Therefore, the entire multiplication takes the form of a *block-by-panel* operation.

2.4 General-Purpose Computing on GPUs

In the early days of General-Purpose Computing on Graphics Processing Units (GPGPU) it was impossible to write efficient, compute-bound matrix multiplication routines due to the lack of developed memory hierarchy on the devices. This changed with the introduction of NVIDIA Compute Unified Data Architecture (CUDA) in 2007, which along other changes brought a fully fledged memory hierarchy introduced to the GPUs. Further, around the same time NVIDIA released their Tesla series products targeted specifically at High Performance Computing (HPC), offering very high single and double precision floating-point performance. Shortly after, in 2008 ATI/AMD released their series of GPUs implementing the non-proprietary OpenCL standard also targeting the HPC sector [2].

GPUs are specialized in compute-intensive, highly parallel computation, as opposed to more general purpose CPUs. They are particularly well suited for stream processing, where the same kernel function can be executed independently on many different data elements in parallel. High arithmetic intensity of GPU computations and high degree of parallelism make it possible to hide memory latency with computation rather than a hierarchy of caches. There are many parameters associated with GPU programming, which need to be carefully chosen to ensure that there is always some computation available to be scheduled on the GPU processing units, while fetching data from memory. The following subsections gives an introduction to the two dominant general-purpose GPU computing platforms: CUDA and OpenCL.

2.4.1 CUDA Programming Model

Compute Unified Data Architecture (CUDA) is a proprietary technology available only on NVIDIA GPUs. It comes with a programming environment which uses C with a minimal set of extensions as a high-level implementation language allowing developers to access CUDA resources. More detail on the language can be found in the CUDA C Programming Guide [15]. CUDA GPUs are characterised by their *Compute Capability.* The GPUs used for the purpose of my investigation (Tesla K40c and GTX 780 Ti) are of Compute Capability 3.5.

Kernels, Blocks and Threads

CUDA developers implement *kernels* – C functions which are executed N times by N different CUDA *threads*. For convenience, CUDA threads are grouped together into one-, two- or three-dimensional structures called *blocks*. Blocks, similarly to threads, are organised into a one-, two- or three-dimensional structure called the *grid*. Each thread within a block can be identified using the *thread index*. The *block index* and the *block dimension* are also accessible to each thread and can be used to compute a global index of each thread within the grid. The number of blocks in the grid is usually dictated by the size of the problem a programmer tries to solve. The number of threads in a block is limited by the amount of resources requested by the kernel.

The number of blocks in the grid and threads in the block used to execute a given kernel is called the *execution configuration*. Carefully selecting values for these parameters can have a large effect on the overall performance of the code.

Platform Model

The CUDA Platform Model is depicted in Figure 2.1. NVIDIA GPUs are built from an array of multithreaded *Streaming Multiprocessors* (SMs). Each SM consists of a large number of CUDA *cores* (192 on the *Kepler* architecture). When a kernel is launched, each block is allocated to a single SM and remains there for its lifetime. Blocks execute concurrently and each SM executes a large number of threads at the same time. Instructions are pipelined to leverage instruction-level parallelism within a single thread. The mapping between the execution model and the platform model is depicted in Figure 2.2.

Threads execute in groups of 32 called *warps*. Multiple warps can execute in parallel on a single SM. All threads in a warp start at the same program address and execute the same instructions, but maintain individual program counters and their execution paths can diverge. This is known as the Single Instruction, Multiple Threads (SIMT) paradigm. In the case when threads diverge, all execution paths are serialised until they converge again. It is very beneficial for performance reasons to avoid divergence by writing code with the minimum number of branching instructions.

The execution context (program counters, registers, etc.) for each warp is stored on-chip for the lifetime of the warp, which allows for warp scheduling to happen at no cost. The number of warps resident on the multiprocessor is limited by the number of registers and shared memory requested by the kernel.

Devices with Compute Capability 3.x have a dedicated L1 cache for each multiprocessor as well as an L2 cache shared between all SMs. Additionally, each multiprocessor has a read-only data cache of 48KB to speed up reads from device memory. Devices with Compute Capability 3.5 can access it directly, while devices with lower Compute

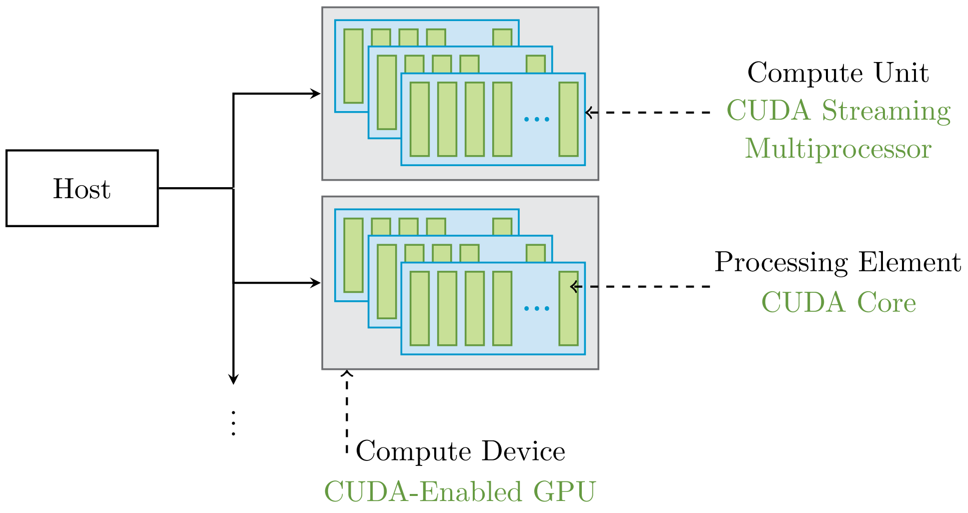


Figure 2.1: CUDA (labelled in green) and OpenCL (labelled in black) platform models. Reproduced from [17].

Capability access it through a texture unit (the cache is hence referred to as the texture cache). The multiprocessors have also a read-only constant cache that is shared by all functional units.

Memory Hierarchy

Global memory resides in device memory and is visible to every thread. It is accessed via 32-, 64-, or 128-byte memory transactions. When a warp of threads reads or writes global memory, the accesses are coalesced into a number of transactions depending on the size of words accessed and the scatter of memory addresses. The less transactions are issued the higher the bandwidth utilisation. Therefore, it is important to maximise coalescing by following the most optimal access patterns and using data types of sizes meeting the alignment requirements. The best performance is achieved when a warp collectively reads consecutive memory locations aligned at a 128-byte boundary. On devices of Compute Capability 3.x global memory reads are not cached in the L1 cache. In addition, global memory can be also cached in a designated read-only data cache.

Local memory is private to each thread and is typically used when register spilling is necessary or when variables are too large to fit in the register space (e.g. arrays). Local memory resides in the device memory and has the same high latency and low bandwidth as global memory. On devices of compute capability 2.x and higher, local memory accesses are always cached in the L1 and L2 level caches. Further, local memory undergoes the same alignment requirements as global memory.

Shared memory has a much lower latency than global memory, because it is located on-chip. It is visible to each thread within a given block. To achieve high bandwidth, shared memory has been divided into equally-sized memory banks, which can be accessed simultaneously by a number of threads. Thus, the most optimal memory access pattern for a warp is when each thread accesses data from a different bank. Otherwise, a bank conflict occurs and the accesses have to be serialised. A special case occurs when all threads in a warp access the same data element, which can be broadcast and accesses do not need to be serialized. There are two addressing modes for the shared memory (64-bit and 32-bit), which allow successive words of 64- or 32-bits to map to successive banks. Shared memory is equivalent to a user-managed cache.

Constant memory is read-only, resides in the device memory and is cached in the *constant cache*. It can be accessed by all threads.

Texture memory is also read-only, resides in the device memory and is cached in the *texture cache*. The texture cache is optimised for 2D spatial locality. It can be accessed by all threads.

2.4.2 OpenCL Programming Model

Open Computing Language (OpenCL) [17] is a framework providing developers a language based on C99 to implement kernels and execute them across heterogeneous systems of CPUs, GPUs, FPGAs and other hardware accelerators. OpenCL is a unified platform for parallel programming on devices such as high-performance servers, personal computers or even mobile phones.

Kernels, Work-Groups and Work-Items

Alike the CUDA platform, OpenCL developers partition their problem into coarse grain sub-problems and implement *kernels* to solve them. Submitting a kernel for execution on an OpenCL device defines an index space s.t. one kernel instance, known as the *work-item*, is executed for each point in this space. Work-items are aggregated into *work-groups*, which correspond to CUDA blocks. Work-items in a given work-group execute concurrently on the processing elements of a single compute unit. Similarly to CUDA, the index space can be 1-, 2-, or 3-dimensional, and work-items can be identified by either their *global ID* or the combination of a *work-group ID* and their *local ID* within the group.

The developer specifies the number of work-items required for the given computation. OpenCL allows the programmer to further define the division of work-items into work-groups (the explicit model – useful when work-items need to share Local Memory) or leave this decision to the OpenCL implementation (the implicit model).

Platform Model

The OpenCL Platform Model (largely similar to the CUDA one as depicted in Figure 2.1) consists of the *host*, which executes the host program according to its native model and submits *commands* (memory transfers, synchronisation barriers or *kernel* launches) to the *OpenCL devices* it is connected to. Each of the compute devices can contain multiple *compute units*, which execute a single stream of instructions as Single Instruction, Multiple Data (SIMD) units. Further, each compute units can have multiple *processing elements*, which can execute as Single Program, Multiple Data (SPMD) units maintaining their own program counter (equivalent to CUDA SIMT). The host maintains a *command-queue* to coordinate execution of kernels on the devices. The mapping of the execution model onto the platform model is shown in Figure 2.2.

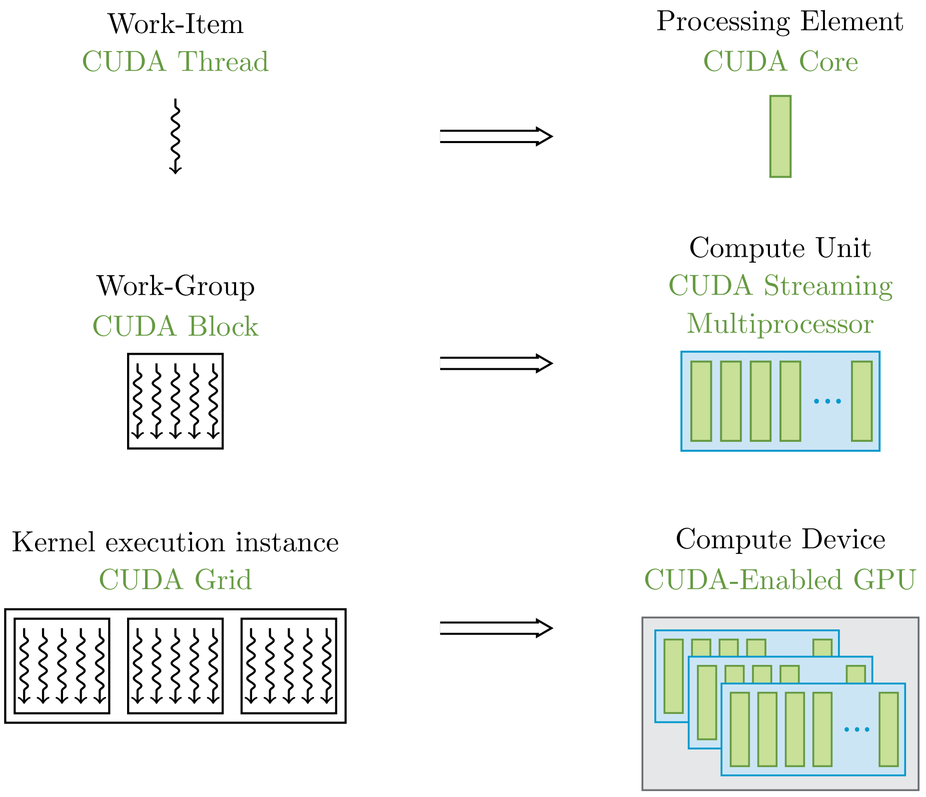


Figure 2.2: Mapping the kernel execution model onto the platform model. CUDA labelled in green and OpenCL labelled in black. Reproduced from [24].

OpenCL is designed for heterogeneous computing and hence does not put any requirements on how the model is implemented. For the purposes of our investigation it is worth taking a look at the AMD *Hawaii* architecture (corresponding to the tested FirePro W9100). The Hawaii devices have 4 Shader Engines of 11 compute units each. Each compute unit consists of 64 shaders (corresponding to the processing elements).

Each compute unit has its dedicated on-chip L1 cache and all 4 Shader Engines share an L2 cache [5], which is largely similar to what CUDA offers as well.

Memory Hierarchy

Global Memory is available to all work-items across all work-groups and corresponds to CUDA global memory. Depending on the capabilities of the device the accesses to this memory may be cached.

Constant Memory remains constant through the kernel execution and can only be allocated by the host. It corresponds to CUDA constant memory and similarly can be accessed by all work-items.

Local Memory is private to each work-group. Depending on the device it might be implemented as dedicated regions of memory or be mapped onto sections of global memory. It corresponds to the CUDA shared memory.

Private Memory is the OpenCL equivalent of CUDA local memory and is private to each work-item.

2.4.3 Basic Performance Optimisation Strategies for GPGPU

Maximising Resource Utilisation. It is important to select the execution parameters in such a way to enable allocation of a large number of CUDA blocks (work-groups) simultaneously on the device. This includes partitioning the workload into enough chunks and splitting each into an appropriate number of threads (work-items) to keep the resources requirements of each block as small as possible. The SMs (compute units) rely on thread-level parallelism to maximise utilisation of their functional units, therefore it is crucial that the scheduler always has a warp ready to execute. The most common reason why a warp is not ready to execute is when its operands need to be fetched from memory and are not yet available. It can also be limited by dependencies between instructions (one thread waiting for another thread to write some data). By ensuring that a large number of blocks can reside simultaneously on each compute unit, we explicitly facilitate latency hiding through providing a large pool of warps ready for execution. Number of registers and the amount of shared memory (local memory) used by each thread (work-item) can also limit the number of warps that can reside simultaneously on a multiprocessor.

Maximising Memory Throughput. The techniques for optimising memory throughput rely on an efficient use of shared memory (local memory in OpenCL), L1/L2 caches, texture cache and constant cache as well as minimising the number of reads from device memory. Further, to fully utilise the available memory bandwidth one needs to coalesce and align all accesses to device memory in order to issue the smallest possible number of memory transactions and reduce the overhead of instruction replays. For CUDA devices with Compute Capability 2.x and higher the same on-chip memory is used for both the shared memory and L1 cache. The proportion of memory dedicated to each space can be configured for each kernel call.

Maximising Instruction Throughput. Instructions throughput can be increased by minimising threads divergence within warps, reducing the number of instructions and by trading precision for speed by using single rather than double precision arithmetic or intrinsic functions (less accurate but faster versions of standard arithmetic functions). In this thesis we intend to investigate both cases of single and double precision, but are not in a position to make an argument whether precision of a fluid flow simulation can be traded off for speed.

2.5 cuBLAS, clBLAS and cuSPARSE

The most recent release of PyFR targets the CUDA and OpenCL platforms and defaults to the use of cuBLAS and clBLAS GEMM for matrix multiplication operations when executing on GPUs. These libraries are widely considered to be the best available GPU BLAS alternatives and hence are the subject of this investigation.

cuBLAS is the highly optimised NVIDIA's dense BLAS library [14], which is the most popular choice for CUDA GPUs. Regretfully, cuBLAS is not open-source and the implementation details are hidden from the developers. However, through profiling one can develop an intuition about the basic techniques used by cuBLAS to provide a fast GEMM implementation. As expected, we notice a high utilisation of shared memory by cuBLAS, which suggests the routine employs some form of tiling to perform the matrix product.

clBLAS is the popular OpenCL dense BLAS implementation [1], which can run across a variety of hardware accelerators. It is open-source and therefore provides the developers with complete information regarding the implementation of the matrix multiplication routine. clBLAS is particular interesting as it can tune its performance to a particular system. The auto-tuning utility comes packaged with the library and works by executing different variants of the GEMM routine in the search for the one that performs best on the given system. In our investigation we have tuned the clBLAS library prior to running the benchmarking suite for all devices.

cuSPARSE is the NVIDIA's sparse version of the BLAS library for CUDA [16]. It is designed to provide the best performance for GEMM in the cases where the matrices are largely sparse. It requires the matrices to be stored in a compressed format such as Compressed Sparse Row (CSR) or Coordinate List (COO), but this does not pose a significant overhead as the matrices in our problem remain constant for the duration of the simulation. One might think that cuSPARSE should be preferred over cuBLAS as the provider of matrix multiplication kernels in PyFR for simulations running on hexahedral and quadrilateral meshes (these element types correspond to sparse operator matrices). However, Georgiou [7] found that the use of cuSPARSE for the case of small matrices alike these used in PyFR delivers worse performance than the dense cuBLAS GEMM. For this reason we will not consider cuSPARSE in this investigation.

Chapter 3

Related Work

General Matrix Multiplication is a well-studied problem and many techniques exist that deliver near peak performance for the typical use cases. We give an overview of these techniques in Section 3.1. It is far less common for someone to challenge the implementation of BLAS libraries in order to deliver higher performance GEMM routines. Our investigation is motivated by a particular application in PyFR, which means that the operator matrices and the type of the matrix product have certain characteristics (size, sparsity, etc.) that are known in advance and can be exploited in order to outperform the state-of-the-art GEMM implementations. Sections 3.1 and 3.2 present some of the most fundamental optimisation techniques underlying the implementation of modern high-performance BLAS libraries on various platforms. Section 3.3 outlines an approach taken in the development of the SD++ solver (the first solver for Flux Reconstruction schemes), where the authors develop bespoke matrix multiplication kernels to grant performance improvements to their application. Section 3.4 summarizes an approach to implementing high-performance, small-scale BLAS by generating code for fixed-size linear algebra expressions, where the matrices in question are also know in advance. Section 3.5 gives examples of other work in the field of linear algebra on the GPU platform, which suggest a series of techniques for achieving high performance matrix multiplication kernels. The methodology used in our investigation will be further discussed in Chapter 4 and the evaluation of our findings in Chapter 5.

3.1 Anatomy of High-Performance Matrix Multiplication

A large amount of work on General Matrix Multiplication (GEMM) routines has been based on the findings of Goto and van de Geijn [8] from 2008. Although their paper is the most applicable to CPUs, it gives some invaluable insights into how matrix product can be efficiently engineered on any platform. The authors illustrate a layered approach

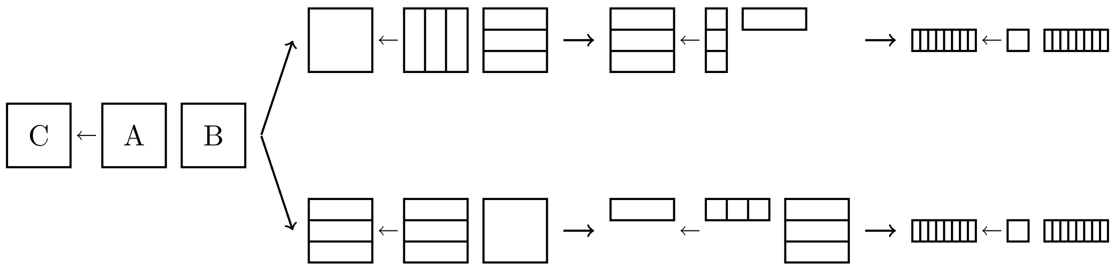


Figure 3.1: The diagram illustrates two (out of six) ways suggested by Goto, how general matrix multiplication can be successively decomposed. The two ways shown in this diagram use block-by-panel type of matrix product at the last level of decomposition. Reproduced from [8].

to implementing GEMM, capable of optimally amortizing the additional cost of moving data between different levels of memory hierarchy. They show how the implementation of matrix multiplication can be decomposed into multiplications with submatrices (known as blocking). The computation is cast into multiple calls to the *inner-kernels*, which perform the multiplication of blocks. The authors identify six inner-kernels that are considered for building blocks for high-performance GEMM, one of which is the block-by-panel type of matrix product. Figure 3.1 illustrates a decomposition of GEMM in two (out of total six) ways suggested by Goto, which utilise block-by-panel matrix product kernels at their lowest levels. The idea presented in Goto's paper is that if the lowest level kernels can attain high performance, then so will the main case of GEMM. Further, other linear algebra operations can often be cast in terms of these special shape matrix multiplication inner-kernels.

In the aforementioned paper the authors explain how the size of the matrices used in the inner-kernels should be chosen to allow for high utilisation of the memory caches. Packing of submatrices into contiguous memory is shown to improve the utilisation of the TLB and serves as a mean for bringing the entries of the matrices into the memory cache. Unfortunatly, these optimisations are of little relevence to the GPU platform as the GPUs do not use virtual memory and the L2 cache is so highly contested by thousands of parallel threads that it is difficult to predict its perfromance. However, the GPUs shared memory (known as local memory for OpenCL) can be used as an explicitly managed cache. Further, tiling for register reuse is also considered in the paper and demonstrated to bring performance gains to the kernels.

The paper by Goto and van de Geijn identifies a series of key factors for achieving high-performance GEMM implementation on any platform and stresses the importance of the six inner-kernels building blocks of GEMM in the implementation of various other routines such as level-3 BLAS.

3.2 Automatically Tuned Linear Algebra Software

Hand-optimised BLAS libraries are expensive and time consuming to produce, hence they only exist where there is a large enough market to justify the costs. To build a high-performance BLAS library a programmer needs to know all the details about the memory hierarchy, cache sizes, functional units, registers, etc. of the targeted platform. Unfortunately, some vendors do not expose these details, making it much harder to write high-performance code.

Clint Whaley and Jack Dongarra in 1998 suggested a solution to this problem – the Automatically Tuned Linear Algebra Software (ATLAS) [21]. The authors suggest to use code generation coupled with timing routines to perform a search for the optimal implementation of GEMM on a given hardware platform. The methodology for building GEMM is similar to the one given by Goto in [4] and described in Section 3.1. Whaley suggests that GEMM routine can be implemented out of smaller inner-kernels, which need to be optimised for a given platform. ATLAS performs the generation of inner-kernels during the library installation process (*auto-tuning*). The process times execution of a number of different kernels to find the one that performs best in the given scenario. As discussed in Section 2.5, clBLAS comes with an auto-tuning software, which is able to adjust the library's performance to the given hardware platform.

According to Whaley, code generation needs to account for numerous parameters such as the blocking factors, loop unrolling depths, software pipelining strategies, loop ordering, register allocations and instruction scheduling. Further, it needs to be aware of the hardware characteristics in order to guide its search through the parameter space. The hardware specification can be given by users or approximated through micro-benchmarking.

The authors identify a set of problems associated with this approach. Firstly, it is impractical for the auto-tuning facility to exhaustively test all kernels in the parameter space, hence the most optimal solution may not always be found and depends on the information the installer has about the hardware characteristics. Also, they stress the problem with executing GEMM on very small problem sizes, where the naive 3-loop implementation can outperform the 'optimised' code. Lastly, the authors discuss the strategies for generating the cleanup code to take care of data falling outside of the tiling boundaries. The size of the tiles is known during code generation, hence it is possible to generate cleanup code for all possible shapes of the tiles with dimensions smaller than the blocking size. However, this approach does not scale well and would produce large binaries. The proposed solution compromises between the generation of a number of optimised routines for assumed shapes of the tiles and a complementing set of generic routines accepting general tile parameters, but exhibiting worse performance.

3.3 SD++ Solver

Castonguay et al. in their paper [4] describe the development of a GPU-enabled compressible viscous flow solver, which utilises the Flux Reconstruction approach. The authors found that custom kernels for matrix multiplications gave 40% performance increase compared to the cuBLAS GEMM library. The proposed solution made use of the texture memory (cached on chip) and shared memory to benefit from data reuse. According to Castonguay the win over cuBLAS can be attributed to the three following factors:

- The usage of custom kernels allowed to reduce the total number of fetches from the global memory. The kernels could have been modified to perform multiple algebraic transformations on the entries of B and C without the need to launch any separate kernels. This, however, did not maintain the GEMM interface.
- Since the size of operator matrices is small and they remain constant throughout the computation, they could have been loaded into the texture memory granting a very fast access time.
- Due to heavy data reuse, elements of B were loaded into shared memory prior to each multiplication. Shared memory is significantly faster than global memory and can act as a fast user-managed cache.

The proposed algorithm worked in the following way. The B matrix was naturally partitioned along its width into cells. Each block of threads was assigned at least one cell. Each thread within a block was responsible for computing an entire row of the output matrix C. As claimed by the authors, this approach, as opposed to an intuitive thread per element of C, increased the register pressure but decreased the number of fetches of A from global memory and also allowed for higher degree of instruction level parallelism.

Two different methods were adopted to perform the multiplication depending on whether the operator matrix was dense or sparse. As an optimisation in the dense case, matrix A was stored in the texture memory in column-major form. This ensured that all threads within a warp accessed sequential locations of the texture memory. In the sparse case, ELLPACK format was used to pack the operator matrices into contiguous memory and allowed to reduce the number of floating-point operations perform by the kernels by eliminating the zero entries.

Castonguay did not investigate the possibility to increase the performance of their kernels through hierarchical tiling. We believe, that tiling the matrix product could allow to reduce the number of registers required by each kernel and hence increase the occupancy, as the demand for resources would drop. This optimisation has the potential to decrease the register pressure while simultaneously keeping the number of memory fetches the same and maintaining the same degree of instruction level parallelism.

3.4 Basic Linear Algebra Compiler

Spampinato and Püschel [19] have identified the need for high-performance, small-scale basic linear algebra computation, which is currently unattainable with the available state-of-the-art vendor BLAS libraries, which perform best with large problem sizes. The authors present the Basic Linear Algebra Compiler (LGen), which takes as input a fixed-size linear algebra expression and outputs a highly optimised `C` function. The code generation consists of three steps. First, a DSL description of the problem is input and a tiling decision (possibly hierarchical) is made based on the sizes of the matrices. The input gets translated into a second DSL, which contains the tiling decisions and makes the access patterns explicit. In step two loop-level optimisations are performed. These optimisations do not require any sophisticated analysis since they utilise the mathematical representation of the computation. Loop-level optimisations employed in LGen include loop merging and exchange. The second DSL representation is next translated into a `C` intermediate representation (C-IR). In the third stage code-level optimisations such as loop unrolling and translation into SSA form are performed. Lastly, the C-IR code is unparsed into `C`. The performance results obtained using the generated code are fed back to the generator, which uses auto-tuning to refine its initial tiling decisions and perhaps produce even faster code. The performance of the code produced by LGen for small matrices is competitive and often better than the performance of other available libraries.

Knowing the size and structure of the matrices in question prior to the computation step directly corresponds to the problem addressed in this thesis. The authors of LGen suggest that exploring the structure of the matrices can have a significant effect on performance, but leave this claim without any empirical evidence. Further, the optimisations employed in LGem are largely designed for the CPU platform and hence cannot be directly ported onto the GPUs. Nevertheless, the authors present a number of insights, which can prove invaluable when building a matrix multiplication kernel generator for our purposes.

For the purposes of the FR schemes and PyFR it is particularly desirable to optimise the sole matrix-matrix multiplication routine, rather than a combination of linear algebra expressions. Nevertheless, as already identified by Castonguay [4], there is scope to combine the applications of the matrix product with some of the point-local operations performed by PyFR during each time step of a simulation. While this might expose an opportunity for numerous software optimisations to be applied and bring performance improvements to the solver, it can have an unwanted effect of largely reducing the clarity of the implementation.

3.5 Other Work

In 2010 Nath et al. [12] identified the necessary factors for improving performance of cuBLAS GEMM routine once the *Fermi* architecture has been introduced. We recognise that our study is being performed on a newer *Kepler* architecture, which differs significantly to the one considered by Nath, hence their suggested implementation may not be faster than the up-to-date version of cuBLAS. Nevertheless, the points made by the authors remain valuable and relevant. Nath suggests that since registers are much faster than shared memory it might be beneficial to tile for them and hence increase the reuse of data that already resides in the fastest available memory. This might be even more so on the newer architecture with a largely increased number of registers available to each thread. The authors also recognise the benefits of using texture memory to store entries of the operator matrix.

Despite solving a different problem, Jhurani et al. [11] also recognise the issue with reaching peak performance using cuBLAS GEMM when multiplying small matrices, due to the reduced reuse of elements once they have been copied form global memory into shared memory. This insight further reinforces the idea that in the case of small operator matrices, the tiling choices are the most crucial and can have a large impact on performance. It is critical to maximise data reused once it is copied into registers. In this investigation we will explore software optimisations techniques, which should realise this objective. This insight further suggests that selecting an appropriate tiling scheme with multiple levels of blocking, can prove invaluable in developing a fast, high-performance matrix multiplication routine for small matrices.

Chapter 4

Methodology

Due to a vast domain of numerical method choices available for each PyFR simulation, which dictate the characteristics of the used operator matrices (see Section 2.3), we require the matrix multiplication kernels to be auto-generated rather than written and optimised by hand. To accomplish this we have developed GiMMiK– an open-source Python library, capable of generating matrix multiplication kernel code for CUDA and OpenCL.

As part of the preliminary analysis we have considered a series of software optimisations, which we speculated would bring performance improvements to our kernels and enable us to improve over the performance of cuBLAS and clBLAS. These optimisations include:

- using the constant memory to store the operator matrix,
- reducing common sub-expressions,
- sparsity elimination, and
- avoidance of the cleanup code.

We have taken a systematic approach to evaluate each of the proposed optimisations in order to find the successful ones and incorporate them into GiMMiK. By design, each of our kernels computes an entire matrix-vector product. Loop unrolling serves as a basis for all of the applied optimisations.

We note that the desired matrix product is of the form

$$C \leftarrow \alpha AB + \beta C$$

and values of α and β need to be handled with care. Generating kernels with embedded values of the operator matrix allows us to pre-multiply them with α to reduce the required number of floating-point operations. A special case occurs when $\beta = 0$, where we do not need to load entries of the output matrix at all. For this reason, each of the experiments mentioned in this thesis is run twice to investigate the effects of our optimisations in both cases when $\beta = 0$ and $\beta \neq 0$.

In Section 4.2 we propose and experimentally evaluate the four aforementioned ways in which we believe our kernels can outperform BLAS GEMM. Subsequently, we pick the successful optimisations and incorporate them into GiMMiK as described in Section 4.3. Section 4.1 describes the experimental setup used to benchmark our kernels.

4.1 Benchmarking Infrastructure

We have developed a C++/Python benchmarking infrastructure in order to evaluate the performance of our bespoke kernels and compare it with cuBLAS and clBLAS GEMM implementations. The set of matrices used for benchmarking were extracted form the PyFR solver for quadrilateral, hexahedral, triangular and tetrahedral meshes across 1–6 orders of accuracy. The full specification of the matrices is available in Appendix A. The memory layout, the operator matrices and the problem size were setup to mirror those in PyFR. All matrices used in benchmarking are stored in a row-major order and are padded to ensure coalesced and aligned memory accesses by all threads. The operand and output matrices were selected to be $50,000$ elements wide to provide a representative problem size.

Throughout the simulation, PyFR keeps the operand matrix in device memory at all times. It is copied onto the device only at the beginning of the simulation, hence, the cost of memory transfers from the host to the device can be neglected for the purpose of our investigation. Further, the time required to generate and compile our bespoke kernels can also be neglected as it is significantly smaller than the time required to complete any meaningful fluid flow simulation.

The timing of the kernels is done in a standard way using CUDA and OpenCL profiling events. Every reported value in this investigation is an average of 30 kernel executions and is reproducible within 2%. The average was taken to reduce the random error in our observations. 30 un-timed runs were executed before each timing run to eliminate bias due to idle under-clocking of the GPUs. The benchmarks were executed on the NVIDIA Tesla K40c and GeForce GTX 780 Ti, which both share the same *Kepler* (compute capability 3.5) architecture, and the AMD FirePro W9100. The full specification of the devices, including the versions of the OpenCL and CUDA runtime is given in Table 4.1. To provide a comprehensive suite of results, each benchmarking run was executed in single and double precision with $\beta = 0$ and $\beta \neq 0$.

	Tesla K40c	**GTX 780 Ti**	**FirePro W9100**
Architecture	Kepler	Kepler	Hawaii
Compute Capability	3.5	3.5	N/A
ECC	ON	N/A	N/A
Peak Memory Bandwidth	288 GB/s	336 GB/s	320 GB/s
Peak Double-Precision	1.43 TFLOPs	210 GFLOPs	2.62 TFLOPs
Peak Single-Precision	4.29 TFLOPs	5.04 TFLOPs	5.24 TFLOPs
CUDA Version	6.0	6.0	N/A
OpenCL Version	2.0	2.0	2.0

Table 4.1: Specification of the experimental hardware.

4.2 Optimisations Assessment

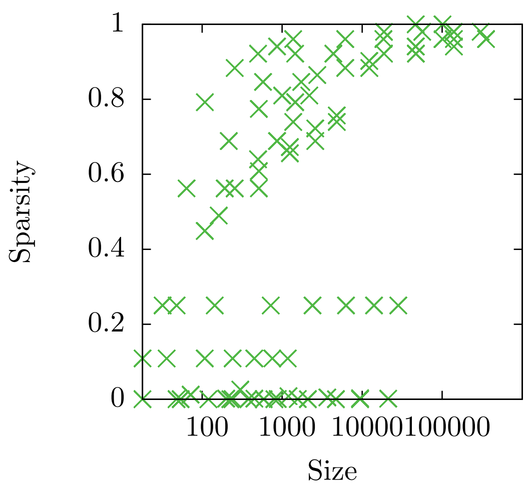
The aim of this section is to systematically evaluate four ways in which we believe our kernels can outperform cuBLAS and clBLAS GEMM in a block-by-panel type of matrix multiplication. For each of these optimisations we have designed an experiment to test whether it can grant the speculated performance benefit. The generated kernels were benchmarked using the infrastructure described in Section 4.1, with the exception that data was only gathered for the CUDA platform on NVIDIA GPUs. Unfortunately, NVIDIA has discontinued support for profiling OpenCL kernels on CUDA GPUs in the latest versions of the toolkit. After the evaluation of the experimental results we are able to select the successful optimisations and incorporate them into our final solution – GiMMiK.

4.2.1 Value Embedding

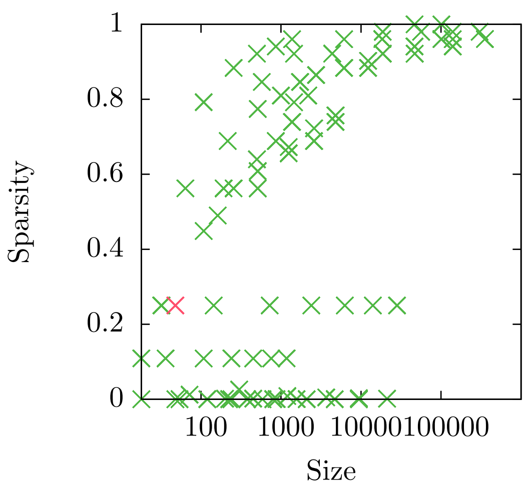
Hypothesis We speculate that encoding values of the operator matrix directly in the kernel code, as opposed to loading them from memory, will result in a performance improvement of our kernels. It is expected to reduce the required number of accesses to global memory and expose the opportunity for the compiler to efficiently reuse values of the operator matrix by retaining them in registers (i.e. eliminate the latency due to memory access entirely). Further, embedding values in the code is realized through storing them in the constant memory. This is advantageous as the compiler has explicit knowledge about the usage pattern of those values and can find an optimal memory storage layout to minimize bank conflicts and best utilise the constant cache. With this approach, for the case of small matrices, the entries of the column of the operand matrix required by each thread are also likely to remain in registers and be reused efficiently while computing the entire column of the output matrix.

Experiment To verify our assumption we generate and benchmark two types of kernels. The first type reads the operator matrix from global memory. On the Kepler architecture global memory loads are not cached in the L1 cache making the *read-only data cache* the only suitable way of caching data. This cache is known as the *texture cache* on older architectures and has to be explicitly managed by the programmer. The CUDA compiler (nvcc) will convert all loads into loads cached in the read-only data cache whenever it can. Hence, to better control experimental variables, we use it explicitly. The second kernel has values embedded in the code and relies on the *constant memory* and the *constant cache* to bring them efficiently into registers.

Discussion Experimental results show that for a majority of benchmark matrices we achieve a significant speedup due to embedding values directly into the kernel code. The cases where the reported execution time of the non-embedding kernels are smaller are mostly within the 2% reproducibility margin and hence considered insignificant. Figure 4.1 illustrates the relative performance of these two types of kernels in double precision for $\beta = 0$. The complementing set of graphs for single precision and $\beta \neq 0$ is available in Figure C.1. Raw experimental data for this experiment is available in Appendix B.



(a) Tesla K40c, $\beta = 0$



(b) GTX 780 Ti, $\beta = 0$

Figure 4.1: Comparison of kernels embedding values the operator matrix in the code and those accessing it through the texture unit for double precision and $\beta = 0$. Positive and neutral effects of value embedding on the set of benchmark matrices are indicated as green.

Unfortunately, NVIDIA does not provide any details on the specification of the constant cache, hence the explanation of these results can be at best speculative. The experimental data suggests that the constant cache offers better performance and locality for the type of memory accesses we require. This is most likely realized through the compiler, which is able to pick the most optimal memory storage layout for the constants in order to avoid memory bank conflicts and fully utilise the constant cache. Profiling reveals that whenever we achieve a particularly large speedup through embedding values in the code, the non-embedding kernels attain a low hit rate in the read-only data cache. This might be due to a particularly unfavourable sparsity pattern, which causes the data to be frequently evicted from the cache. To neutralise the effects of the particular sparsity patterns on the performance of the cache, we could attempt to pack the operator matrix into a contiguous chunk of memory, as suggested by Goto [8]. However, the experimental data for purely dense matrices shows that even when the matrix is stored in a contiguous block of memory the performance of the constant memory/cache is superior to the performance of the read-only data cache.

4.2.2 Common Sub-Expression Elimination

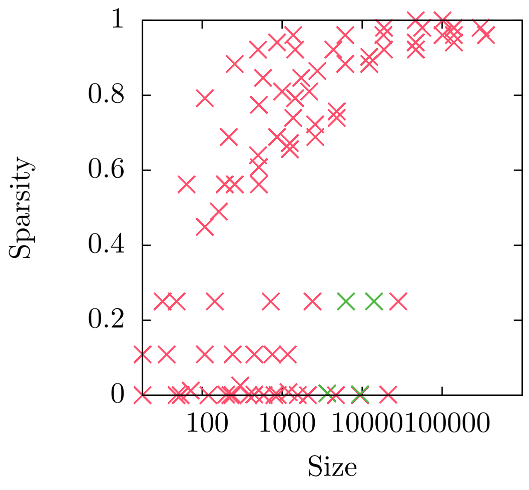
Hypothesis Due to the mathematical formulation of the solution points in various element types, values along the rows of the operator matrix occasionally repeat. This exposes the opportunity for us to reduce common sub-expressions and save on the number of floating-point operations required to compute the matrix product. This can be achieved by summing up elements of the operand matrix corresponding to the repeated values prior to multiplication by the common term. This is illustrated by the following example:

$$\begin{aligned} c_{ij} &= \dots + a \times b_{xj} + \dots + a \times b_{yj} + \dots \\ &= \dots + a \times (b_{xj} + b_{yj}) + \dots \end{aligned}$$

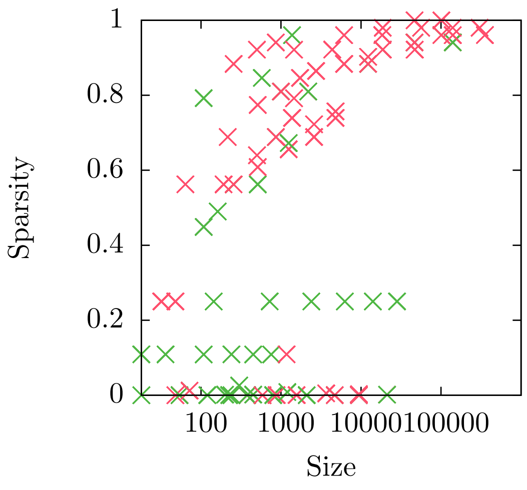
which shows how one multiplication by a common term can be eliminated. The reduction of common sub-expression can bring further performance gains to our kernels.

Experiment To examine whether common sub-expression reduction is beneficial we benchmark two types of kernels, which embed all values from the operator matrix directly in the code and eliminate all multiplications by zeros. Only the second kernel reduces common sub-expressions by summing up entries of B prior to the multiplication by the value from A.

Discussion Experimental data revealed that common sub-expression elimination did not bring the expected effect of improving performance of our kernels. The majority of the kernels carrying out the elimination performs worse or within the 2% reproducibility margin of those that do not. Figure 4.2 illustrates the relative performance of these two types of kernels in double precision for $\beta = 0$. The complementing set of graphs for single precision and $\beta \neq 0$ is available in Figure C.2. Raw experimental data for this experiment is available in Appendix B.



(a) Tesla K40c, $\beta = 0$



(b) GTX 780 Ti, $\beta = 0$

Figure 4.2: Comparison of kernels eliminating common sub-expressions from the operator matrix and those performing no such optimisation for double precision and $\beta = 0$. Positive effects of common sub-expression elimination on the set of benchmark matrices are indicated as green.

When explicitly eliminating floating-point multiplications from the kernel by combining common sub-terms together we trade-off the possibility for the compiler to efficiently reuse elements of the operand matrix. To illustrate this, consider the following operator matrix and an arbitrary constant a:

$$A = \begin{pmatrix} \multicolumn{4}{c}{\dots\dots\dots\dots} \\ 0 & a & 0 & a \\ \multicolumn{4}{c}{\dots\dots\dots\dots} \\ a & a & 0 & 0 \\ \multicolumn{4}{c}{\dots\dots\dots\dots} \end{pmatrix}$$

When eliminating common sub-expressions we would generate the following two sub-terms when computing the *col* column of the output:

```
subterm_10 = b[1 * bstride + col] + b[3 * bstride + col]
subterm_20 = b[0 * bstride + col] + b[1 * bstride + col]
```

The two sums have a common term `b[1 * bstride + col]`, which will be loaded from memory twice, unless it is found in the cache or the compiler kept it in a register when the second sum is computed. Profiling has revealed that reduction of common sub-expressions increased the memory traffic and the number of global memory load and store instructions executed by our kernels. This suggests that the compiler does not change the order in which these sums are computed to increase the likelihood of hitting in the cache and neither does it retain sub-expressions in registers for later reuse. Further, we note an increased number of subterms generated for kernels with reduced common sub-expressions, which increases the register pressure and causes the kernels to demand more resources from the GPU. This, again, has a negative impact on performance and further explains why this optimisation is not a successful one.

However, Figures 4.2 and C.2 show that on the GTX 780 Ti a significantly larger fraction of the kernels benefit from common sub-expression elimination. This is the case as the card offers higher memory bandwidth than the Tesla K40c, hence it is more forgiving of the extra memory loads this optimisation generates. Further, in the case of double precision, the performance cap placed on the floating-point rate means that any reduction of the number of executed arithmetic operations has a potential to bring large performance gains.

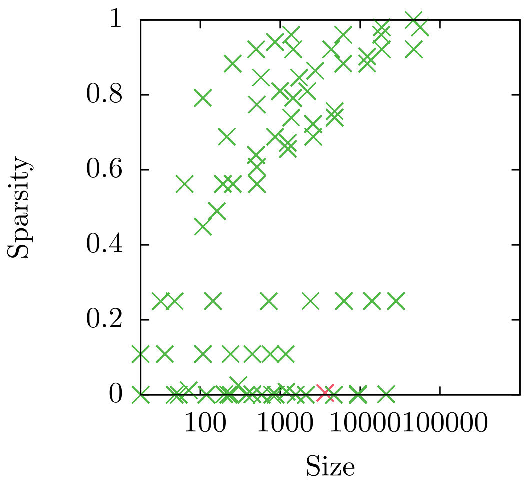
4.2.3 Sparsity Elimination

Hypothesis Thanks to full unrolling we can eliminate all sparsity from the operator matrix by considering only the non-zero entries for the multiplication. In PyFR, quadrilateral and hexahedral element meshes correspond to matrices with large sparsity factors due to the tensor-product formulation of the solution points. We hypothesise that while BLAS GEMM is typically limited by the floating-point performance of the device, decreasing the number of required arithmetic operations (combined with an efficient memory access pattern) can decrease the running time of our kernels.

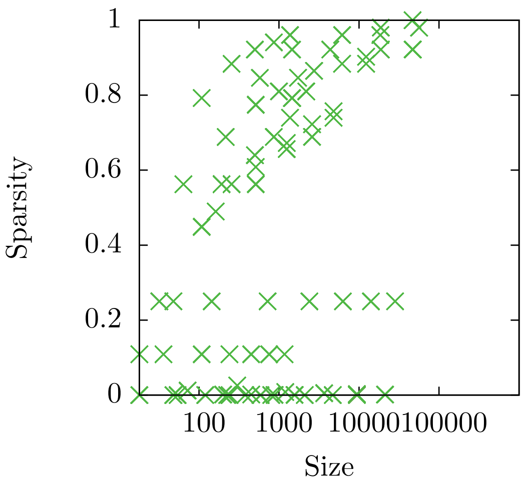
Experiment To verify the effectiveness of sparsity elimination we compare two types of kernels, which embed all values from the operator matrix directly in the code. All multiplications by zeros were eliminated from the second kernel only. Both types of kernels are benchmarked using the earlier mentioned infrastructure to verify the effectiveness of the proposed optimisation.

Discussion In the case of sparse matrices, elimination of zero entries allows us to greatly decrease the number of floating-point operations performed by the kernel. Figure 4.3 illustrates the effectiveness of sparsity elimination in double precision for $\beta = 0$. The complementing set of graphs for single precision and $\beta \neq 0$ is available in Figure C.3. We note that this optimisation has no effect on dense kernels. Raw experimental data for this experiment is available in Appendix B.

This optimisation increases the utilisation of the available memory bandwidth and allows our kernels to execute significantly faster. Profiling has revealed that through the



(a) Tesla K40c, $\beta = 0$



(b) GTX 780 Ti, $\beta = 0$

Figure 4.3: Comparison of kernels eliminating sparsity from the operator matrix and those performing all the multiplications by zeros for double precision and $\beta = 0$. Positive and neutral effects of sparsity elimination on the set of benchmark matrices are indicated as green.

removal of unnecessary floating-point operations our kernels become fully bound by the available memory-bandwidth in both cases of single and double precision. Further, some of our benchmark matrices contain whole columns of zeros, which effectively means that the dimensionality of these matrices decreases and entire rows of the operand matrix do not need to be loaded from memory at all. This not only further decreases the number of arithmetic operations but also reduces the amount of memory that has to be read by our kernels. Naturally, fully dense cases of triangular and tetrahedral element matrices do not benefit from this optimisation. Raw experimental data indicates an existence of a strong positive correlation between the size and sparsity factor of the matrices and the speedup achieved through the removal of multiplications by zero.

4.2.4 Cleanup Code

Hypothesis Lastly, we recognise that the tiling scheme used by cuBLAS and clBLAS might be a limiting factor. GEMM implementations in BLAS libraries often utilise some form of tiling to efficiently reuse data from the cache, registers or – in the case of GPUs – shared memory. Poor tiling choices result in the need to execute cleanup code over the elements of the matrix that fall outside of the tile boundaries (as illustrated in Figure 4.4). This cleanup code is known to be poorly optimised. For the case of small operator matrices such as these used in PyFR, the performance penalty due to poor tiling choices can be particularly significant. Our fully unrolled kernels do not incur any of these costs.

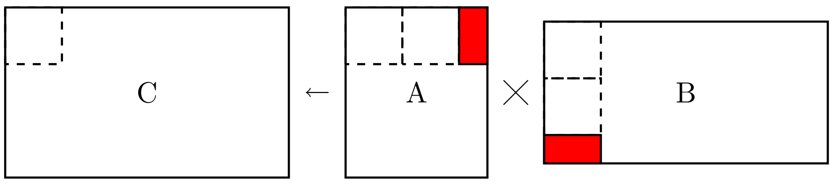
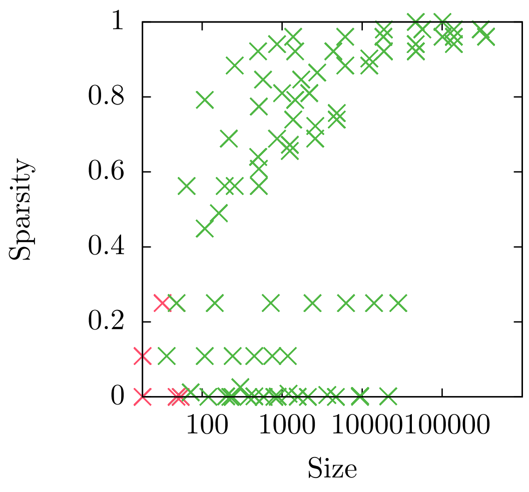


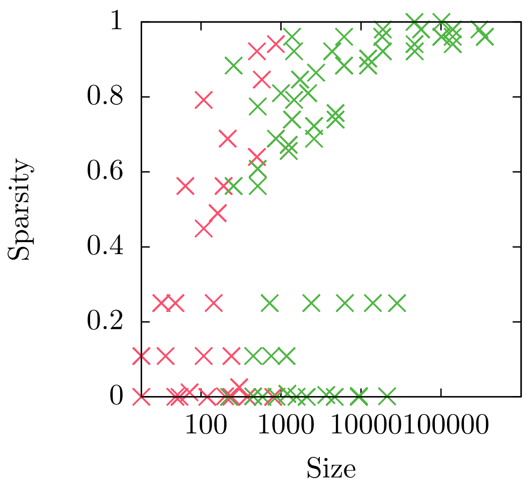
Figure 4.4: Diagram of a blocked matrix product. The red area represents partial tiles, whose dimensions are not a multiple of the blocking factor. These partial tiles are multiplied using the *cleanup code*.

Experiment To examine what effects the cleanup code in cuBLAS has on performance, we benchmark fully unrolled kernels with sparsity eliminated and values of the operator matrix embedded in the code, against the BLAS GEMM and against a naive 3-loop implementation of the matrix product. The naive implementation serves as a reference to determine the extent to which performance of BLAS is dominated by the cleanup code.

Discussion Figure 4.5 illustrates the performance of cuBLAS GEMM and the naive 3-loop implementation in double precision for $\beta = 0$. The complementing set of graphs for single precision and $\beta \neq 0$ is available in Figure C.4 and all experimental data for this experiment is available in Appendix B. We see that for a large fraction of the smaller benchmark matrices the performance of cuBLAS GEMM is often worse than that of a naive matrix product. The analysis of the experimental data has confirmed our conjecture that performance of BLAS GEMM is heavily affected by the poorly optimised cleanup code for cases of small matrices.



(a) Tesla K40c, $\beta = 0$



(b) GTX 780 Ti, $\beta = 0$

Figure 4.5: Comparison of NVIDIA cuBLAS and the naive 3-loop matrix multiplication kernel for double precision and $\beta = 0$. Cases when cuBLAS performs better are indicated as green.

Additionally, we observe that our bespoke kernels are able to achieve very impressive speedups over BLAS for the smallest of the benchmarked matrices (see Chapter 5). This provides further evidence for the hypothesis that the cleanup due to inefficient tiling in cuBLAS and clBLAS GEMM for small matrix sizes results in a large overhead and hence allows our bespoke kernels to perform significantly better.

4.3 GiMMiK

Having performed the experiments described in Section 4.2 we are in a position to generate highly performant kernels for use in the PyFR solver. We have incorporated the successful optimisations into GiMMiK. We conclude that to achieve the best performance of our kernels we will eliminate sparsity from the operator matrix to reduce the number of floating-point operations. Through the preliminary analysis we found that this optimisation was highly beneficial for the matrices with high sparsity factors (corresponding to hexahedral and quadrilateral meshes). Further, GiMMiK will embed all non-zero values directly in the kernel code to benefit from the fast constant cache and compiler optimisations. This optimisation is expected to increase the performance of kernels for all element types. Lastly, we avoid the need to execute any cleanup code through loop unrolling, which shows to be beneficial especially for the smallest matrices (low orders of accuracy) across all element types. GiMMiK's kernels will not reduce common sub-expressions as this optimisation was found to have a negative effect on their performance due to the increased number of memory loads necessary to compute the subterms and an increased register pressure due to larger number of temporary variables needed in the code.

GiMMiK was built as a stand-alone Python package with a simple interface (see Figure 4.6), which can be readily installed and used across a number of systems. It incorporates all of the successful optimisations discussed in this thesis and turns them on by default. Common sub-expression elimination can also be applied on demand, but as discussed previously, is not expected to benefit the performance of our kernels. Figure 4.7 shows a sample code generated by GiMMiK for the CUDA platform for a given operator matrix and $\beta = 0$.

One of the design goals was to make GiMMiK's kernels interchangeable with BLAS GEMM, however, the very nature of some of the applied optimisation meant that we would not be able to maintain the same interface. GiMMiK includes support for α and β coefficients as well as allows the matrices to be strided in memory. The only difference between GiMMiK and BLAS GEMM is that the later allows the A and B matrices to be optionally transposed inside the routine.

Chapter 5 aims to present a final set of benchmarking results obtained through the

```python
import gimmik.generator as gen
from gimmik.platform import Platform

...

# Generate kernel
kernel = gen.generateKernel(data,
                            alpha=2.0,
                            beta=3.0,
                            double=True,                # Precision
                            reduced=False,              # CSE
                            platform=Platform.OPENCL)   # Platform

```

Figure 4.6: GiMMiK's interface.

$$A = \begin{bmatrix} 0.0 & 0.0 & 0.59097691 \\ 0.63448574 & 0.0 & 0.0 \\ 0.0 & 0.71191878 & 0.95941663 \end{bmatrix}$$

(a) Operator matrix input to GiMMiK.

```c
__global__ void
gimmik_mm(const double* __restrict__ b,
          double* __restrict__ c,
          const int width,
          const int bstride,
          const int cstride)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    if (index < width)
    {
        const double *b_local = b + index;
        double *c_local = c + index;

        const double subterm_0 = b_local[2 * bstride];
        const double subterm_1 = b_local[0 * bstride];
        const double subterm_2 = b_local[1 * bstride];

        c_local[0 * cstride] = 0.5909769053580467 * subterm_0;
        c_local[1 * cstride] = 0.6344857400767476 * subterm_1;
        c_local[2 * cstride] = 0.9594166286064713 * subterm_0
                             + 0.7119187815275971 * subterm_2;
    }
}

```

(b) Kernel code generated by GiMMiK.

Figure 4.7: Sample kernel code generated by GiMMiK for the CUDA platform for the given operator matrix and $\beta = 0$.

use of GiMMiK to generate bespoke matrix multiplication kernels for a set of PyFR matrices. We will present the final speedups our kernels achieve over cuBLAS and clBLAS. Further, we will critically assess their performance in terms of the percentage of peak capabilities of the hardware they are able to utilise. Lastly, we will plug GiMMiK into PyFR to investigate the absolute performance improvements to the solver attainable through the use of our package as a matrix multiplication kernel provider for fluid flow simulations.

Chapter 5

Evaluation

At this stage we have systematically evaluated a series of software optimisations and incorporated the successful ones into GiMMiK. As the next step we use the same experimental setup as described in Section 4.1 to evaluate the final performance of our bespoke kernels. All details on the benchmarked operator matrices are available in Appendix A. All experimental results obtained for individual matrices across all element types are available in Appendix D and are discussed in Section 5.1. Further, in Section 5.2 we assess the quality of our solution by taking a look at how much of the peak performance of the device our kernels are able to achieve. As the final step in Section 5.4 we explore the performance benefits the use of GiMMiK grants to our motivating application in PyFR and evaluate the limitations of our solution in Section 5.5.

5.1 Kernels Benchmarking

We find that our bespoke CUDA kernels are able to outperform cuBLAS GEMM in nearly all cases for quadrilateral, hexahedral and triangular element matrices in double and single precision on both the GTX 780 Ti and Tesla K40c with $\beta = 0$ as well as $\beta \neq 0$. These matrices are either sparse or small and dense. In the case of larger tetrahedral matrices the library implementation proves superior. This is expected as these matrices are dense and sufficiently large for cuBLAS to perform well. Additionally, in the case of single precision we find a small number of large hexahedral matrices, where cuBLAS is slightly more performant than our kernels, as the library GEMM implementation can fully utilise the inherently higher single precision floating-point performance of the devices without it being a limiting factor.

Similarly, we find that our OpenCL kernels are able to outperform clBLAS GEMM in all cases for quadrilateral, hexahedral and triangular element matrices in double and single precision with $\beta = 0$ as well as $\beta \neq 0$ on both the GTX 780 Ti and Tesla K40c. We have only benchmarked our kernels in double precision and $\beta = 0$ on the FirePro W9100 and found the a significantly smaller portion of the benchmarked matrices was able to achieve a speedup over clBLAS GEMM. On the FirePro W9100 some of the larger sparse matrices and mid-range dense matrices, which showed performance improvements on the CUDA platform, proved to perform worse than clBLAS.

The top plots in Figures 5.1 to 5.4 illustrate the achieved speedups across a variety of matrix sizes and sparsity patterns corresponding to the benchmark matrices from PyFR for 1–6 orders of accuracy and $\beta = 0$ on the CUDA platform. The complementing set of plots for the OpenCL platform and $\beta \neq 0$ is available in Appendix E. We conclude that sparse matrices and particularly small matrices benefit the most from optimisations employed by GiMMiK. This is expected while GiMMiK reduces the number of floating-point operations required to compute the product as well as avoids the poorly optimised cleanup code, which dominates performance of GEMM for small matrices. Further, we observe cases of matrices with the exact same size and sparsity, that achieve different speedups. These matrices differ in their width and height, which has an effect on how well BLAS performs, due to the inefficiencies in the tiling choices it is able to make for these particular dimensions. It has little effect on performance of our kernels. In the case of double precision on the GTX 780 Ti we note particularly impressive speedups due to the artificial cap imposed by NVIDIA on the peak double precision floating-point rate on the consumer-grade card. Through the use of GiMMiK we are able to significantly reduce the number of floating-point operations required and hence better utilise the available resources.

Lastly, we want to compare the relative performance of the CUDA and OpenCL kernels when executed on the same platform (GTX 780 Ti and Tesla K40c). We observe that in double precision, on both the GTX 780 Ti and Tesla K40c our CUDA kernels generally perform better than OpenCL kernels for a majority of the benchmark matrices, apart from cases for larger dense matrices corresponding to higher-orders tetrahedral meshes. This is, however, within the performance region where cuBLAS and clBLAS operate better than our bespoke kernels. In single precision our CUDA kernels perform worse than OpenCL for the majority of larger dense and sparse matrices, corresponding to tetrahedral and hexahedral meshes respectively. The lack of ability to profile OpenCL kernels on CUDA devices limits our understanding of the performance differences between these two platforms. We speculate, that because NVIDIA's OpenCL implementation is 32-bit (CUDA is 64-bit) and hence uses less registers to perform address calculations and to store pointers, it might be able to reduce the register pressure affecting performance of our kernels for large matrices. Through this observation we develop a general belief that the CUDA kernels should be favoured on the NVIDIA's platform.

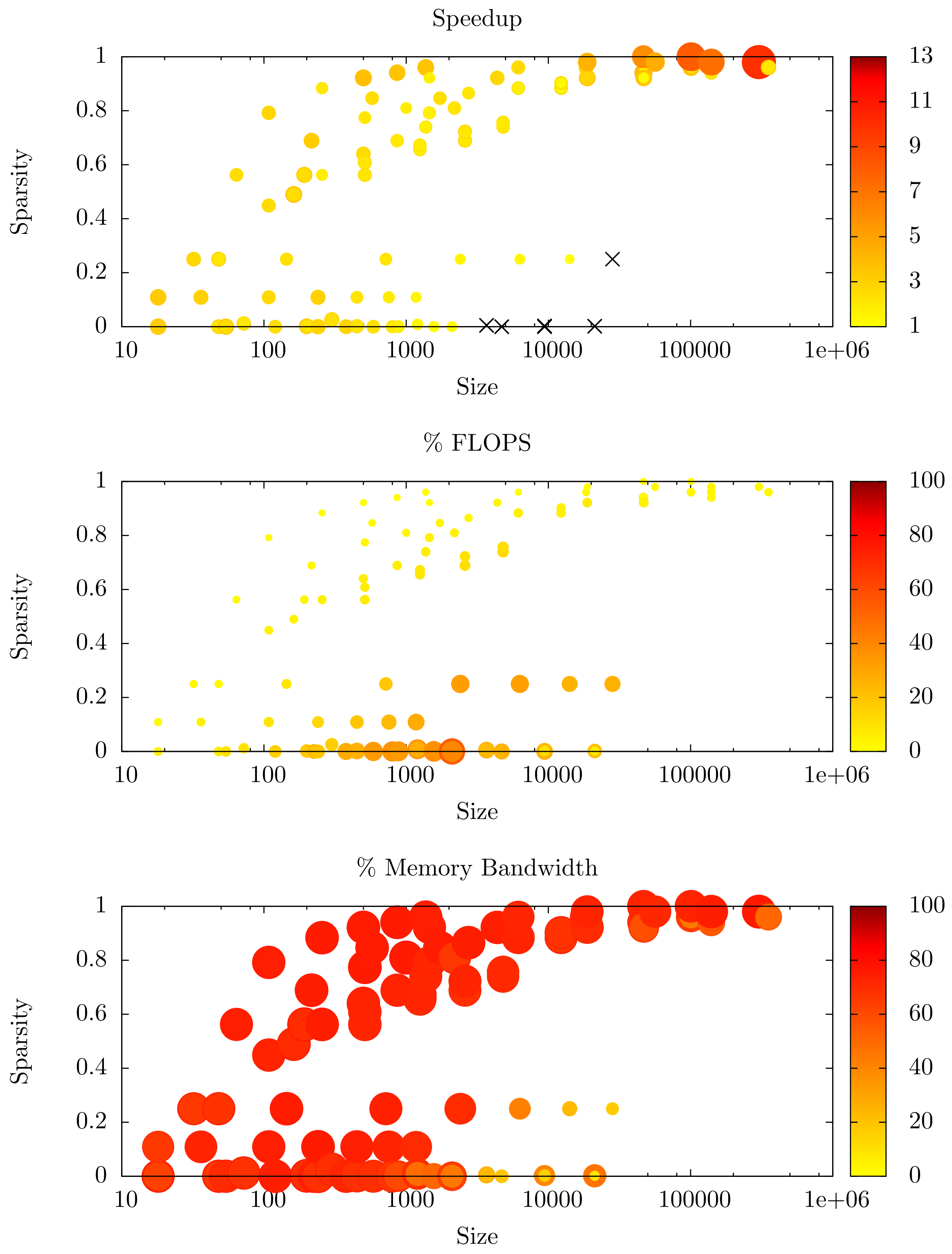


Figure 5.1: Plots illustrating the speedup of GiMMiK's CUDA kernels over cuBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision $\beta = 0$ on Tesla K40c.

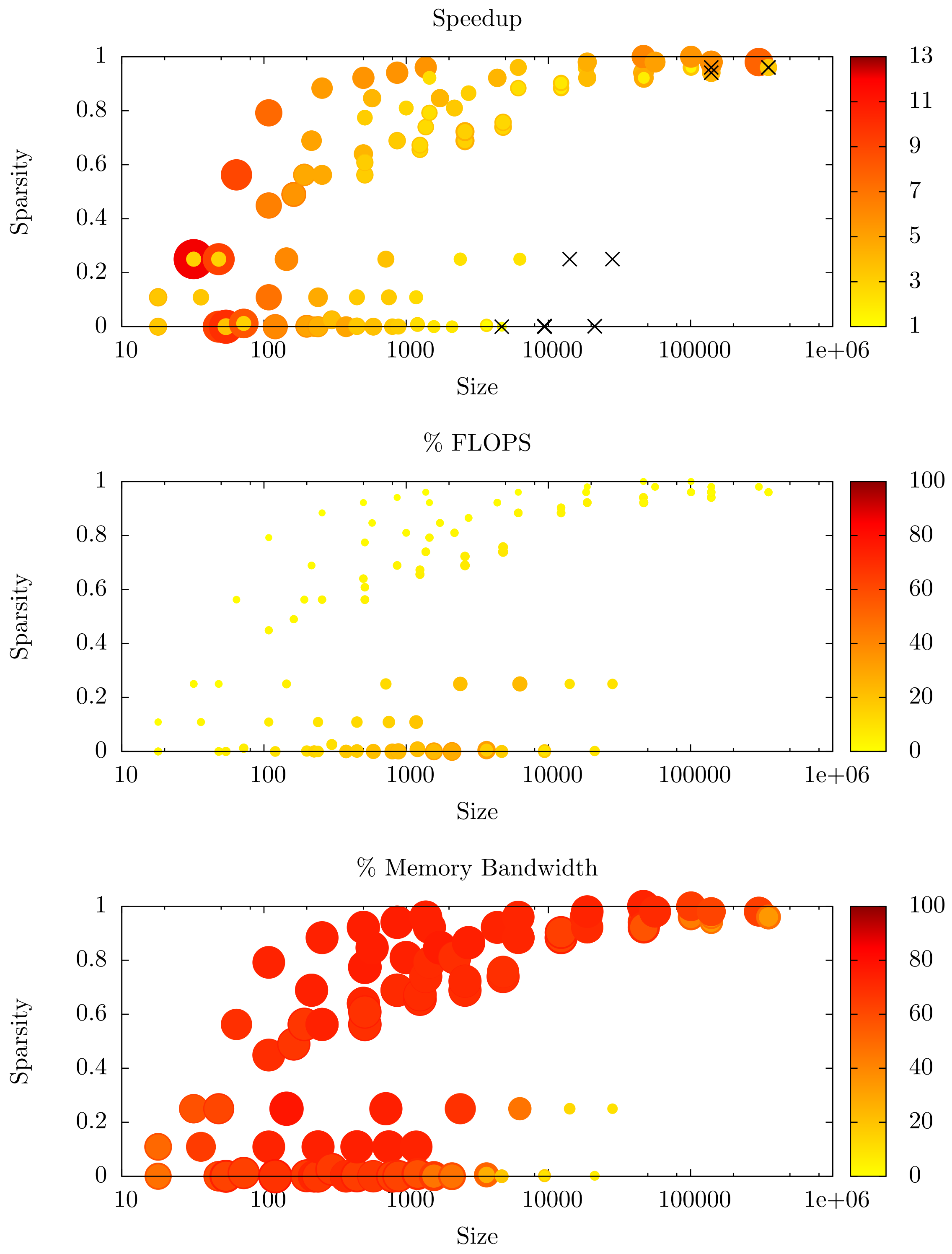


Figure 5.2: Plots illustrating the speedup of GiMMiK's CUDA kernels over cuBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision $\beta = 0$ on Tesla K40c.

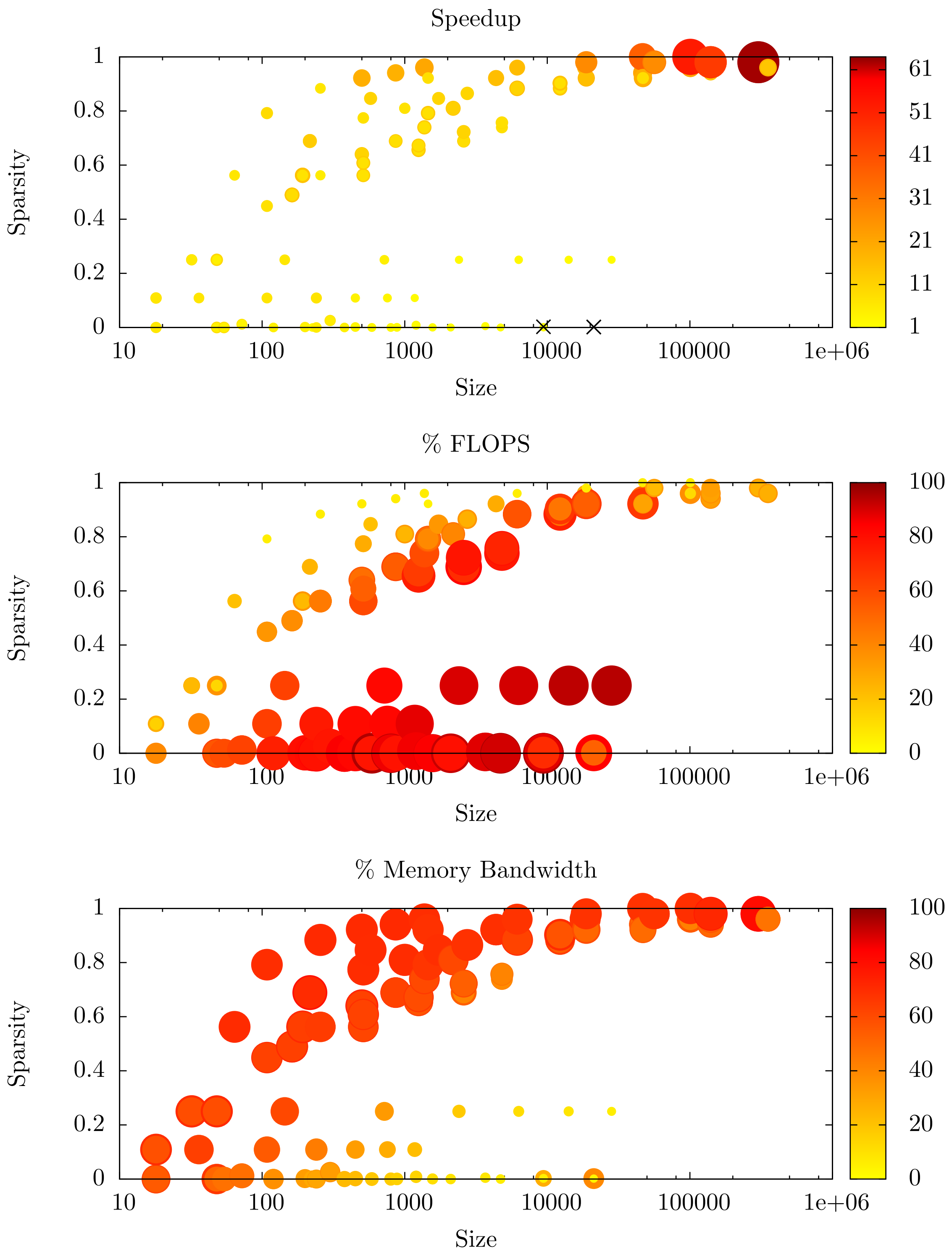


Figure 5.3: Plots illustrating the speedup of GiMMiK's CUDA kernels over cuBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision $\beta = 0$ on GTX 780 Ti.

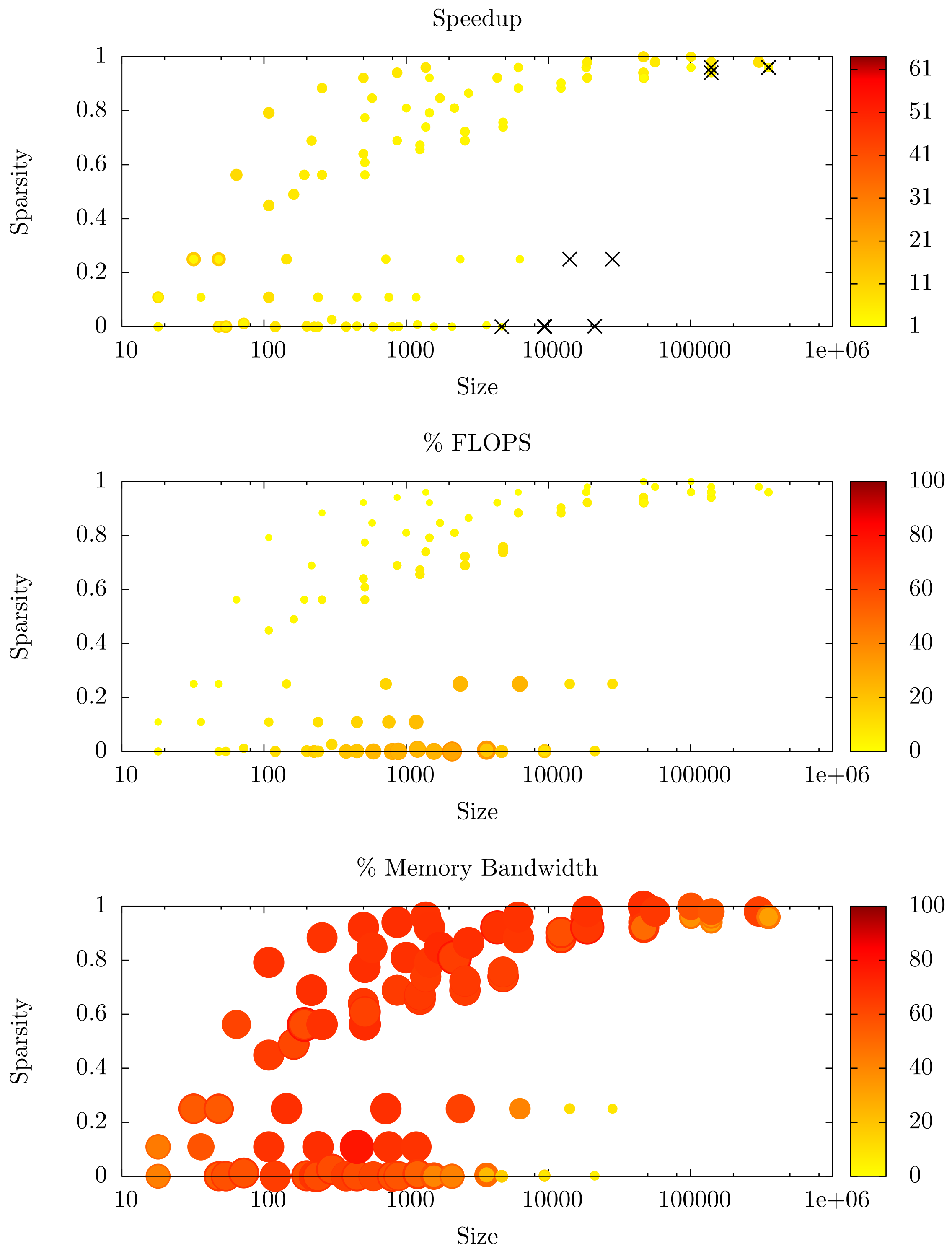


Figure 5.4: Plots illustrating the speedup of GiMMiK's CUDA kernels over cuBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision $\beta = 0$ on GTX 780 Ti.

5.2 Quality Assessment

Performance of BLAS libraries is typically bound by the floating-point capabilities of the hardware [12], because GEMM routines do not exploit any sparsity or redundancy in the data. To assess the quality of GiMMiK we want to empirically determine the performance limiting factor for each of the generated kernels. Firstly, we calculate the ratio of the peak floating-point performance and peak memory bandwidth for each device using values reported by its vendor. Next, for each kernel on each device we calculate the ratio of the achieved floating-point rate and memory bandwidth. We compare the two ratios to determine the limiting factor for the performance of the kernel:

$$a = \frac{achieved\_flops}{achieved\_bandwidth} \qquad\qquad p = \frac{peak\_flops}{peak\_bandwidth}$$

such that when $a < p$ the kernel is bound by the available memory bandwidth, else by the floating-point performance of the device.

The achieved floating-point rate is defined in terms of the number of floating-point operations needed to perform the matrix product and the time taken to execute the kernel. We use the NVIDIA profiler (nvprof) to count the number of floating-point operations executed by our kernels. This is more accurate than computing this number algebraically, due to the reduction of the number of operations resulting from the compiler optimisations and elimination of sparsity.

The achieved memory bandwidth is computed in terms of the time taken to execute the matrix product and the total amount of memory needed to be read and written by the kernel. Similarly to the floating-point rate, the achieved memory bandwidth was obtained through profiling. We note that the metric reported by the compiler is inclusive of any traffic to and from local memory.

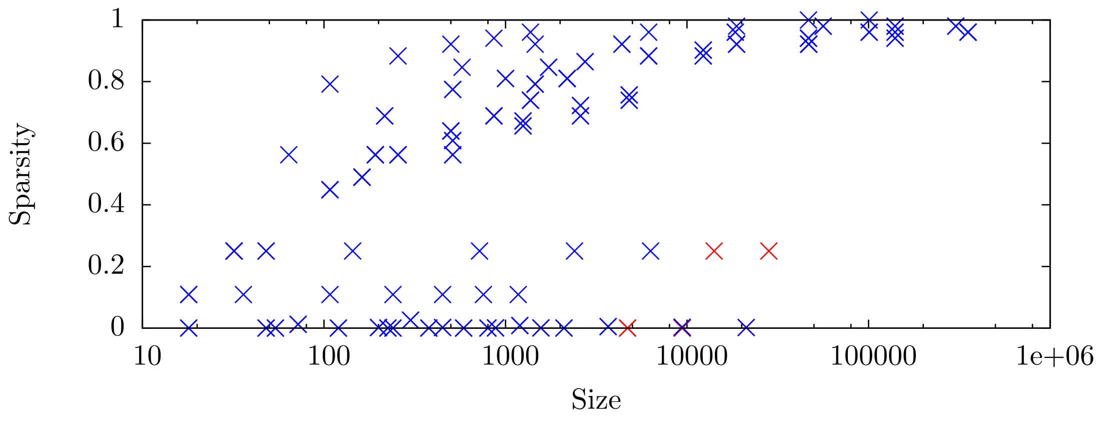
Profiling of our kernels has revealed that through the applied optimisations all kernels for sparse matrices (hexahedral and quadrilateral meshes) become bound by the available memory bandwidth. Small dense kernels do not require enough floating-point operations to utilise a large percentage of the available floating-point rate and hence are also bound by the available memory bandwidth. Kernels for larger sized dense matrices, are limited by the floating-point performance of the device. These results hold in both single and double precision across all devices on the CUDA platform. Due to the artificially reduced double precision floating-point rate on the GTX 780 Ti, a larger portion of our dense kernels turn out to be bound by the available floating-point rate on that device. We also find a small number of larger matrices, which despite being predominantly sparse, do not contain enough zeros to overcome the performance cap. Figure 5.5 illustrates these findings for kernels in double precision and $\beta = 0$; complementing figures for single precision are available in Appendix G. Unfortunately, we were unable to profile the OpenCL kernels on CUDA GPUs as NVIDIA discontinued support for OpenCL profiling in the latests releases of their toolkit. Nevertheless, we expect the results to conform to the described trends.

The bottom two plots in Figures 5.1 to 5.4 and Appendix E illustrate the achieved percentage of the peak floating-point rate and memory bandwidth by GiMMiK's kernels for various size and sparsity patterns corresponding to the benchmarked PyFR matrices. By the analysis of these plots we notice that the utilisation of the available memory bandwidth dramatically drops for large dense matrices (also for very large sparse matrices) and results in performance degradation of GiMMiK's kernels up to the point where they perform worse than library GEMM. Further, not unexpectedly, dense matrices achieve higher utilisation of the available floating-point rate than sparse matrices. We observe a significantly higher percentage utilisation of the double precision floating-point rate on the GTX 780 Ti due to the artificial cap imposed by NVIDIA on their consumer-grade hardware.

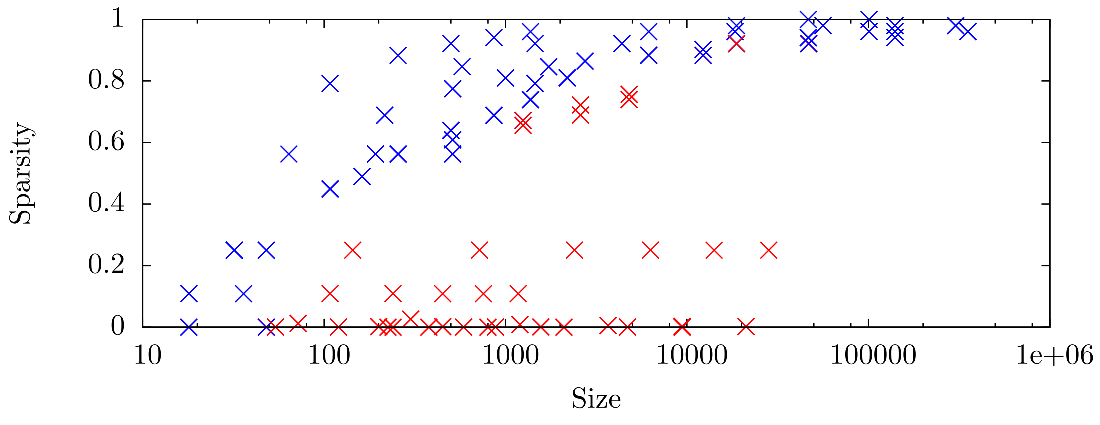
5.3 Hardware Performance Assessment

Having developed an understanding of the factors limiting the performance of our kernels, we can now evaluate the performance of the three pieces of hardware we have used for our investigation. However, comparing the absolute performance of the three devices with each other would not be very meaningful, as they all have different specifications. One could normalize the achieved results by the market price of each GPU and its running cost in order to find what hardware should be used to build a cost-optimal system for a given application. This investigation, however, is beyond the scope of this thesis.

Nevertheless, it is insightful to compare the case of the consumer-grade and the industry-grade cards together. Our study shows that the largest speedups are obtained for the consumer-grade GPU for both cases of dense and sparse matrices. This is expected for the memory bandwidth bound kernels as the GTX 780 Ti offers higher memory bandwidth than Tesla K40c. Further, in the dense cases the artificial FLOPS limit in double precision on the GTX 780 Ti poses a big disadvantage to BLAS GEMM, and hence results in a much larger speedup achieved by our kernels. In terms of the absolute performance, GTX 780 Ti executes GiMMiK's kernels faster than the Tesla K40c in all cases of sparse matrices in double precision, which are bound by the available memory bandwidth. Further, the GTX 780 Ti is superior in all cases in single precision due to the higher hardware specification.



(a) Tesla K40c, $\beta = 0$



(b) GTX 780 Ti, $\beta = 0$

Figure 5.5: Memory bandwidth bound (blue) and floating-point rate bound (red) double precision, $\beta = 0$ kernels for a set of benchmark matrices.

5.4 Performance Improvements of PyFR

In order to investigate the effects of the applied optimisations on the performance of PyFR as a whole, we will first take a look at the way the matrix multiplication kernels are applied to the data at each time step of the simulation (we will only consider the the CUDA platform in this study). For this purpose we will use the same notation as in Section 2.2 to refere to the operator matrices. Each of the $M132$, $M3$, $M460$ and $M6$ is applied to the data once, while $M0$ is applied 3 times on a 2D mesh and 4 times on a 3D mesh. In PyFR all applications of $M0$, $M132$ and $M460$ use $\beta = 0$, while the applications of $M3$ and $M6$ use $\beta = 1$. As explained previously in Section 2.3, the operator matrices exhibit different characteristics for quadrilateral, hexahedral, triangular and tetrahedral elements, hence, there is a need for a distinct set of kernels for each element type present in the mesh. We use the experimental data obtained through benchmarking of the individual matrices to construct a series of stacked plots, from which we can obtain an expected speedup of each of the matrix multiplication steps executed by PyFR for each element type. The calculated results and plots for the third order of accuracy are available in Appendix F. This however, does not translate directly to the final speedup of a simulation as it does not account for a series of other operations performed by the solver (e.g. point-local transformations, writing solution files, checking for NaNs, etc.). The percentage of the time PyFR spends on matrix multiplication increases with the desired order of accuracy and amounts to approximately 56%, 66% and 81% for the second, third and fourth order simulations [23]. Hence, we expect to achieve larger speedups for higher orders of accuracy through the use of GiMMiK.

The aforementioned stacked plots also include theoretical limits under which GEMM cannot perform – they are defined in terms of the minimum number of floating-point operations and amount of memory traffic any dense matrix multiplication routine has to perform. In the case where the operator matrix A has dimensions $M \times K$ and the operand matrix B has dimensions $K \times N$, GEMM needs to perform approximately $2 \times M \times N \times K$ floating-point operations. It also needs to write $M \times N$ elements of C and read $K \times N$ elements of B. In the case when $\beta \neq 0$ we also need to read C and perform an additional $M \times N$ floating-point multiplications with the β coefficient. We can now combine the theoretical amount of memory that needs to be moved and the number of floating-point operations that need to be performed with the peak memory bandwidth and floating-point rate of each device to compute the theoretical lower bounds on the performance of any GEMM routine.

By analysing the stacked plots and the data in Table F.1 we see that through the use of GiMMiK for the sparse cases we are sometimes able to perform the matrix multiplication step of a PyFR simulation under the theoretical limits of the hardware. We observe this phenomenon for a majority of kernels executed in double precision on the GTX 780 Ti, but also a small number of kernels on the Tesla K40c. This is possible when the multiplication step is composed of individual operator matrices that contain whole columns of zeros, which effectively means that the dimensionality of these matrices decreases and entire rows of the operand matrix do not need to be loaded from memory at all. It also decreases the amount of floating-point operations required to be executed by the kernel.

To investigate the effective performance improvement that our bespoke kernels bring to PyFR we have executed an example real-world simulation of a compressible unsteady flow over a cylinder. For our purpose we have not dealt with the physical feasibility of the solution, but solely with the performance of the solver. For full details regarding the setup of the simulation please refer to the original paper on PyFR [23]. The simulation was executed over an unstructured mesh of $46,610$ hexahedral elements depicted in Figure 5.6 on the Tesla K40c and GTX 780 Ti GPUs. Achieved speedups of PyFR for this simulation for 1–6 orders of accuracy and across our two devices on the CUDA platform are summarized in Table 5.1. Isosurfaces of density captured during this simulation executed using GiMMiK as a matrix multiplication kernel provider for PyFR are illustrated in Figure 5.7.

Through the use of bespoke matrix multiplication kernels, we are able to grant significant performance benefits to PyFR. We note speedups between 1.35 and 1.72 for an example fluid flow PyFR simulation executed in double precision on a tetrahedral mesh across 1–6 orders of accuracy on a single Tesla K40c. In practical terms it means that if one had a simulation which takes 1000 hours to execute in the fourth order of accuracy on a tetrahedral mesh in double precision, they could expect to execute it in as little as 580 hours using GiMMiK. Our results have the potential to influence the numerical method choices made for various types of fluid simulations, while the performance in-

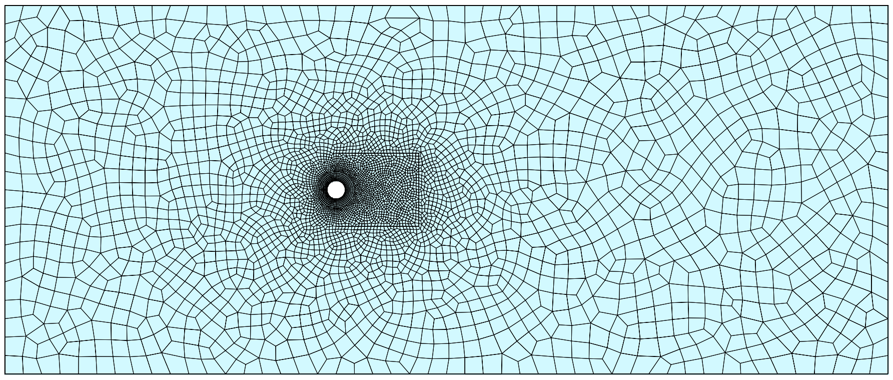


Figure 5.6: Cross section in the x–y plane of the unstructured cylinder mesh used to execute the simulation of an unsteady flow over a cylinder.

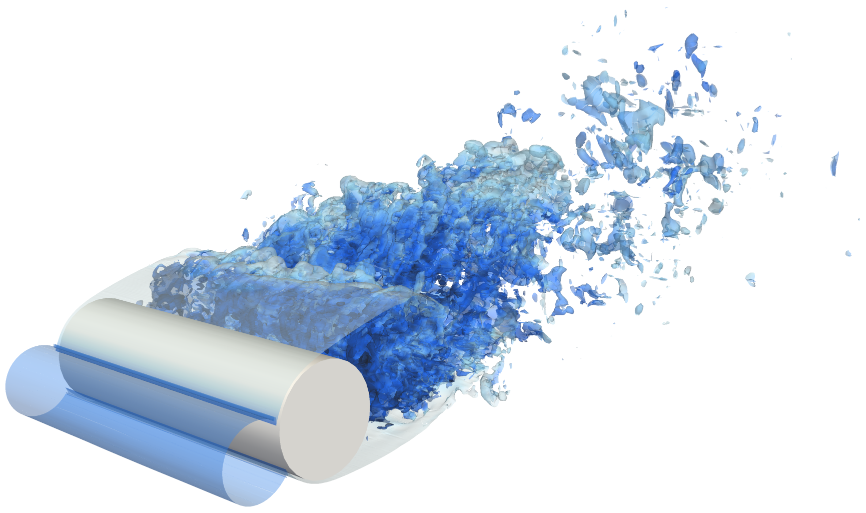


Figure 5.7: Isosurfaces of density captured during a simulation of an unsteady flow over a cylinder executed with PyFR using GiMMiK as the matrix multiplication kernel provider.

Order	Tesla K40c		GTX 780 Ti	
	single	double	single	double
1	2.14	1.51	2.29	6.99
2	1.66	1.35	1.77	6.94
3	1.50	1.40	1.61	7.14
4	1.69	1.72	1.85	10.38
5	1.66	1.58	1.78	8.95
6	1.54	1.64	1.60	10.11

Table 5.1: Achieved speedups of a PyFR simulation of an unsteady flow over a cylinder across 1–6 orders of accuracy in single and double precision.

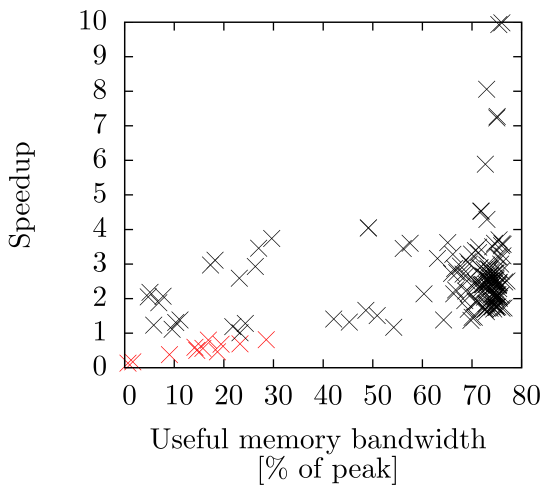
crease, to some extent, can allow for use of higher order meshes and finer time steps. This would improve the physical properties of the simulation and increase the quality of the produced solutions. Unfortunately, this is not very straightforward. Increasing the order of accuracy of a simulation requires other parameters to change as well (e.g. the granularity of the time step needs to be finer for higher orders of accuracy). In order to develop a good understanding of how the numerical method choices can be influenced by the performance increase, we would need to perform a rigorous experimental investigation accounting for the physical soundness and feasibility of the produced solutions. This, however, is beyond the scope of this project.

5.5 Limitations

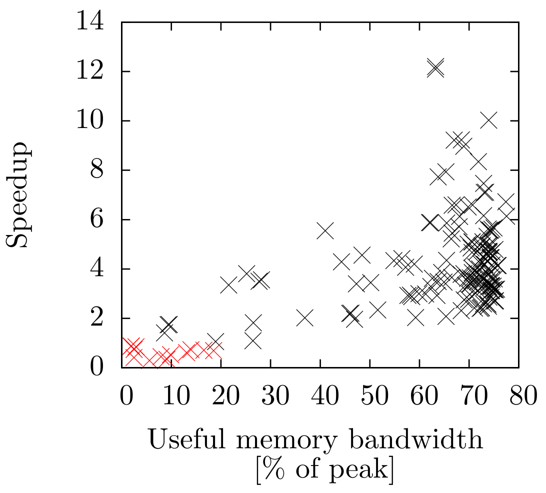
Figures 5.1 to 5.4 and Appendix E identify a set of GiMMiK's kernels which do not benefit from our proposed optimisations and fail to achieve any speedup over cuBLAS GEMM. These kernels correspond to larger sized dense benchmark matrices in cases of both single and double precision. Additionally, in single precision we note decreased performance of our kernels for large sparse matrices. Through profiling we observe that kernels which fail to achieve a speedup, attempt to utilise a very large (often the maximum available) number of registers, which decreases their occupancy on the multiprocessors and increases register pressure due to a large number of temporary variables used. Further, CUDA GPUs employ register scoreboarding to facilitate out-of-order execution [13], which promotes the usage of a larger number of registers to avoid data hazards. Increased register pressure can lead to a large amount of register spillage into memory. In the case of GiMMiK's kernels for larger matrices this spillage is so severe that the data cannot be retained in the L1 or even L2 level caches. As a consequence of this, it needs to be written and read from local memory. This increases memory traffic, decreases occupancy and as a consequence decreases performance of our kernels. It also explains why the achieved fraction of the peak memory bandwidth and floating-point rate of our kernels for large matrices drops. We believe that to overcome this limitation we can tile the matrix product in such a way to bring the register pressure down and hence achieve higher occupancy and reduce spillage of registers into local memory.

Figure 5.8 further illustrates the effects of the achieved memory bandwidth and the amount of data spillage into local memory on the speedups of GiMMiK over cuBLAS. It shows that the speedups over cuBLAS shrink as the observed *useful* memory bandwidth decreases (device memory bandwidth exclusive of traffic due to local memory)[1]. In the case of double precision on the GTX 780 Ti we see that the observed useful memory bandwidth is often lower than in the remaining cases due to the cap placed on the floating-point rate, which becomes a significant factor affecting the performance of our kernels. From the plots we see that the the smallest speedups are obtained by kernels which utilise a smaller percentage of the available memory bandwidth.

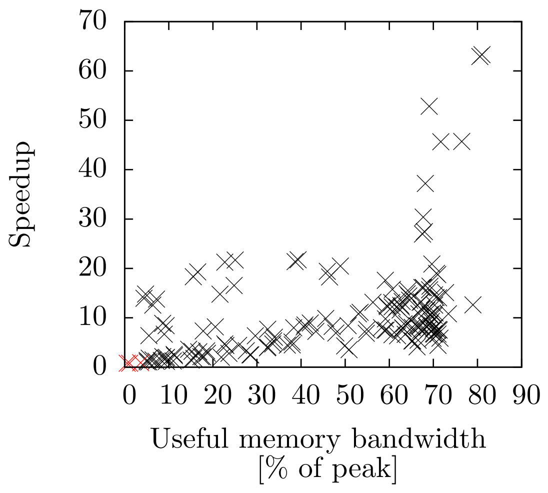
[1]Useful memory bandwidth was calculated using the total device memory bandwidth and local memory overhead as reported by nvprof.



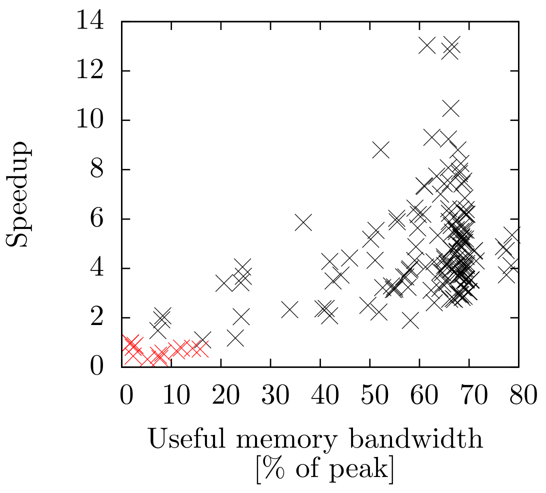
(a) Double precision, Tesla K40c, $\beta = 0$



(b) Single precision, Tesla K40c, $\beta = 0$

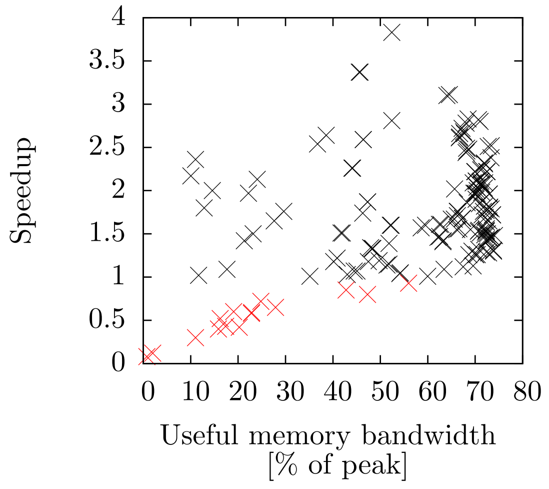


(c) Double precision, GTX 780 Ti, $\beta = 0$

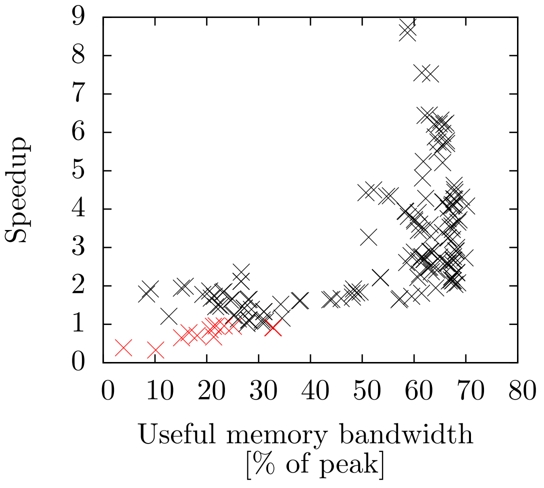


(d) Single precision, GTX 780 Ti, $\beta = 0$

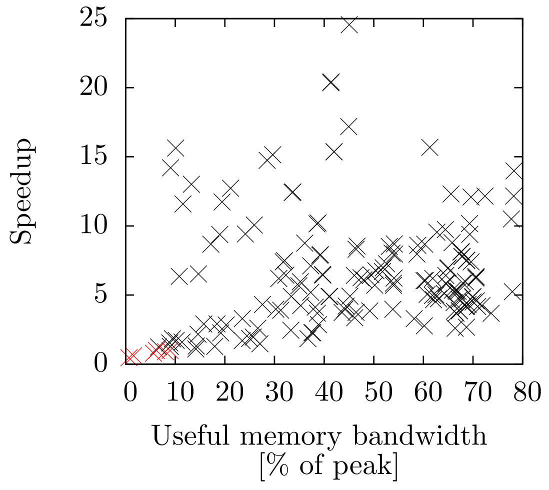
Figure 5.8: Plots of speedup against useful memory bandwidth (exclusive of local memory traffic) expressed as a percentage of the peak memory bandwidth for kernels generated for the set of benchmark matrices and $\beta = 0$. Speedups less than 1 are denoted red.



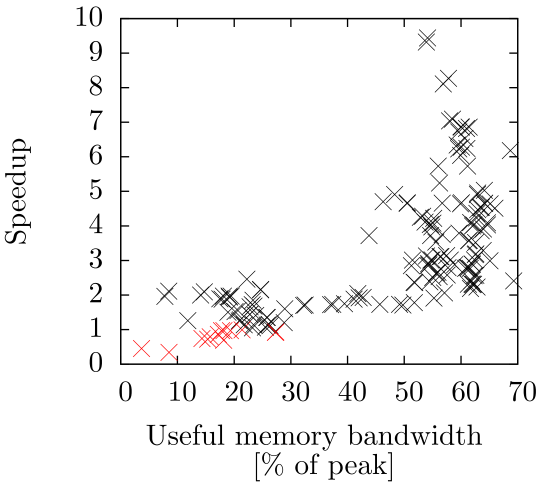
(e) Double precision, Tesla K40c, $\beta \neq 0$



(f) Single precision, Tesla K40c, $\beta \neq 0$



(g) Double precision, GTX 780 Ti, $\beta \neq 0$



(h) Single precision, GTX 780 Ti, $\beta \neq 0$

Figure 5.8: Plots of speedup against useful memory bandwidth (exclusive of local memory traffic) expressed as a percentage of the peak memory bandwidth for kernels generated for the set of benchmark matrices and $\beta \neq 0$. Speedups less than 1 are denoted red.

Chapter 6

Conclusions & Future Work

The aim of this chapter is to give a summary of our findings and address the initial list of objectives set for this investigation. In Section 6.2 we discuss the scope for further investigation of the performance improvements GiMMiK can deliver to other applications, we suggest a series of software optimisations our bespoke kernels could benefit from and attempt to address the limitations of our solution described in Section 5.5.

6.1 Summary

The aim of this project was to investigate the performance improvement over the state-of-the-art BLAS GEMM achievable through the use of bespoke matrix multiplication kernels for a block-by-panel type of matrix multiplication characteristic to PyFR. In Chapter 2 we provide details about the context of our study and describe the implementation platform our our choice – CUDA and OpenCL. Subsequently, in Chapter 3 we give an overview of the work that lays foundations to many high-performance implementations of GEMM in software. We also investigate other research in the field of small-scale linear algebra and draw insights from the described techniques to apply them in our investigation. In Chapter 4 we detail the methodology used in this study. We describe the benchmarking infrastructure used and provide a systematic evaluation of a series of software optimisations in order to find the successful ones and incorporate them into GiMMiK. Chapter 5 provides an empirical evaluation of the performance GiMMiK's kernels are able to achieve for the set of benchmark matrices extracted from the PyFR solver. Lastly, we plug GiMMiK into PyFR and investigate the performance benefits it grants for an example fluid flow simulation.

In this thesis we have made the following contributions:

- We have demonstrated that through generation of bespoke matrix multiplication kernels with a prior knowledge of the operator matrix we are able to outperform highly optimised state-of-the-art cuBLAS and clBLAS libraries and grant speedups of up to 9.98 (12.20) times on the Tesla K40c and 63.30 (13.07) times on the GTX 780 Ti in double (single) precision for individual PyFR matrices.

- We present a series of software optimisation techniques which allow us to achieve this result for the block-by-panel type of matrix multiplication. We evaluate each of the proposed optimisations in a systematic way by benchmarking it across a wide variety of matrices with different sizes and sparsity patterns in order to assess its usefulness and success.

- We have presented GiMMiK– an open-source Python library for generating highly performant matrix multiplication kernel code for the CUDA and OpenCL platforms, which incorporates all of the successful optimisation discussed in this thesis. The software is available for download at `https://github.com/bartwozniak/GiMMiK`.

- The generated kernels in our proposed solution consist of a fully unrolled matrix inner product with reduced number of floating-point operations through the removal of sparsity. Further, our kernels embed the operator matrix directly in the code to benefit from the constant cache and compiler optimisations. This design allows us to avoid execution of any cleanup code due to inadequate tiling choices and to efficiently handle values of the α and β coefficients. GiMMiK, by default, does not attempt to reduce common sub-expressions because this optimisation was found to grant negative effects on performance.

- We show how, through the use of bespoke matrix multiplication kernels, we are able to grant significant performance benefits to applications within the area of CFD. We report speedups up to 1.72 (2.14) for an example fluid flow PyFR simulation executed on a tetrahedral mesh in double (single) precision on a single Tesla K40c. These results can influence the numerical method choices made for various types of fluid simulations, while the performance increase, to some extent, can allow for use of higher order meshes and finer time steps, improving the physical properties and increasing the quality of the results of the simulation.

A general paper *"GiMMiK - Generating Bespoke Matrix Multiplication Kernels for Various Hardware Accelerators; Applications in High-Order Computational Fluid Dynamics"* by Bartosz D. Wozniak, Freddie D. Witherden, Peter E. Vincent and Paul H. J. Kelly has been prepared for submission to the *Computer Physics Communications* journal, based on the findings in this report. It is available upon request. We believe that numerous other applications, possibly outside of the area of CFD, can also benefit from the use of GiMMiK as the kernel provider for matrix multiplication operations performed on the CUDA and OpenCL platforms. Further, we are confident that the methodology applied in this study can give a valuable insight into efficient implementations of small-scale linear algebra kernels on GPUs.

6.2 Future Work

CPU implementation As the initial steps in our investigation we have considered the CPU as an implementation platform for our kernels, however, keeping in mind the particular application of our kernels in Flux Reconstruction schemes, the GPU platform became the subject of our study. Nevertheless, there also exists scope for performance improvement over the CPU BLAS libraries such as *ATLAS*, *GOTOBLAS* or *Intel MKL*. Further, GiMMiK already supports the OpenCL platform and one might be interested to evaluate its performance across a set of hardware accelerators such as the *Intel Xeon Phi*.

Instruction Reordering In Section 5.5 we have already described the problem our kernels encounter for larger matrix sizes. The dramatically increasing number of registers requested by our kernels leads to large amounts of memory spillage into local memory and harms the overall performance. The Kepler architecture statically schedules instructions within the warp [18]; it is the NVIDIA's compiler that attempts to find the most efficient schedule for instructions. From the disassembled *cubin* files we see that the *nvcc* compiler is not capable of efficiently interleaving the arithmetic operations performed by our kernels with the memory writes and reads. We can reorder instructions explicitly in the code and reuse temporaries to aid the compiler in allocating a smaller number of registers. Further, we speculate that an appropriate tiling scheme could prove successful in reducing the register pressure as well.

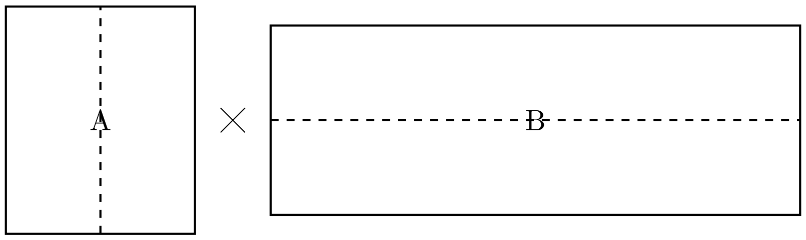


Figure 6.1: Proposed simple split-in-two tiling scheme.

Register Tiling Through Graph Partitioning We have also suggested the use of tiling to overcome the problems introduced by reduction of common sub-expressions. As discussed in Section 3.1 blocking is one of the fundamental techniques used by many BLAS implementations in order to increase data reuse from the memory caches and registers. Simple tiling, such as splitting the operand matrix into upper and lower halves (see Figure 6.1) would reduce the number of subterms in the generated code and hence decrease the number of registers our kernels require. However, this scheme would not solve the problem with repeated loads of elements of B and might even reduce the scope for reduction of common sub-expressions. To aid this, we could use graph partitioning algorithms (e.g. hypergraph partitioning) to find the most efficient and balanced split of the entries in the operator matrix across two separate threads. Consider a matrix with the sparsity pattern depicted in Figure 6.2. An efficient way to tile this product would be to assign columns $0, 1, 5, 6$ to one thread and $2, 3, 4$ to another. Such an allocation would allow us to load each entry of the B operand matrix exactly once, reduce common sub-expressions and leave each thread with a similar workload in terms of the amount of memory reads, writes and the number of arithmetic operations the thread needs to perform.

$$\begin{array}{c|ccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & \times & \cdot & \cdot & \cdot & \cdot & \cdot & \times \\
1 & \cdot & \times & \cdot & \cdot & \cdot & \times & \cdot \\
2 & \cdot & \cdot & \times & \times & \times & \cdot & \cdot \\
3 & \cdot & \cdot & \times & \times & \times & \cdot & \cdot \\
4 & \cdot & \times & \cdot & \cdot & \cdot & \cdot & \times
\end{array}$$

Figure 6.2: Example sparsity pattern suitable for graph partitioning tiling.

Arithmetic Operations Aggregation We have described in Section 3.3 the approach Castonguay et al. took in the development of their bespoke kernels for use in the SD++ solver. They were able to combine multiple arithmetic transformations into a single kernel to increase data reuse and reduce the required memory traffic. Such an optimisation is very dependent on the particular application one would like to use GiMMiK for. In order to further investigate the available performance improvements to PyFR we could attempt a similar optimisation, but would simultaneously need to weigh the benefits it brings and the costs due to reduced clarity of the codebase. Further, to make this optimisation more generic, we could use techniques such as *delayed evaluation* to first build up an entire arithmetic transformation and then optimise it and generate highly performant code. We could look into some of the techniques used by LGen (see Section 3.4) to realise this proposition.

In-Place Multiplication The design of our kernels allows for an in-place matrix multiplication i.e. $C \leftarrow \alpha AC + \beta C$. A discussion with the authors of PyFR suggested that this might be useful to reduce the memory usage of the solver, while such operations are not uncommon and BLAS GEMM cannot perform in-place multiplication. Further, such type of matrix product can be considered an optimisation for the CPU platform, as it reduces the cache footprint. To our knowledge, this path of an investigation has not been considered before and might be an interesting one to explore.

Using GiMMiK to Build GEMM Goto and van de Geijn [8] in their paper illustrate how block-by-panel matrix multiplication kernels can be used as a building block for a general matrix product where all of the used matrices are large. We have summarized this claim in Section 3.1. To use GiMMiK as the inner-kernel of GEMM we would need to give up its highly specialized nature. In order to maintain the high performance of our kernels we could embed the sparsity pattern in the kernel, but rely on the read-only data cache to maintain entries of the operator matrix quickly accessible. In Section 4.2 we have demonstrated that embedding values of the operator matrix directly in the code and utilising the constant cache to bring them into registers is not the game-changing optimisation used by our kernels, hence the performance penalty should not be severe. This would not make our GEMM implementation completely general but suitable for use in applications repeatedly multiplying operator matrices with the same sparsity pattern.

Commodity Hardware Lastly, we have demonstrated that we are able to achieve impressive speedups over cuBLAS GEMM on commodity hardware. In order to answer the question whether consumer-grade GPUs could be used to perform high-order fluid flow simulations instead of expensive industry-grade cards, we would need to perform a detailed benchmarking of a number of simulations, taking into account their physical feasibility and accuracy, and considering the exploitation costs of the hardware.

Bibliography

[1] clBLAS: OpenCL BLAS. `http://clmathlibraries.github.io/clBLAS/`, August 2013. [Online; accessed 11-May-2014].

[2] History of CUDA, OpenCL, and the GPGPU. `https://www.olcf.ornl.gov/kb_articles/history-of-the-gpgpu/`, November 2013. [Online; accessed 23-May-2014].

[3] P. Castonguay, P. E. Vincent, and A. Jameson. A New Class of High-Order Energy Stable Flux Reconstruction Schemes for Triangular Elements. *J. Sci. Comput.*, 51(1):224–256, April 2012.

[4] P. Castonguay, D. Williams, P.E. Vincent, M. López, and A. Jameson. On the Development of a High-Order, Multi-GPU Enabled, Compressible Viscous Flow Solver for Mixed Unstructured Grids. June 2011.

[5] Charlie Demerjian. A long look at AMD's Hawaii and Volcanic Islands architecture. `http://semiaccurate.com/2013/10/23/long-look-amds-hawaii-volcanic-islands-architecture/`, October 2013. [Online; accessed 23-May-2014].

[6] P. Estival and L. Giraud. Performance and accuracy of the matrix multiplication routines : CUBLAS on NVIDIA Tesla versus MKL and ATLAS on Intel Nehalem. Technical report, March 2010.

[7] M. Georgiou. Developing Optimized Matrix Multiplication Kernels for CPU and GPU Platforms: Application to High Order CFD. Master's thesis, Imperial College of Science, Technology and Medicine, September 2013.

[8] K. Goto and R.A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[9] H.T. Huynh. A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods. *AIAA paper*, 4079, 2007.

[10] H.T. Huynh. A Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin for Diffusion. *AIAA paper*, 403, 2009.

[11] C. Jhurani and P. Mullowney. A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices. *CoRR*, abs/1304.7053, 2013.

[12] R Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[13] NVIDIA. Kepler GK110 whitepaper. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012. [Online; accessed 24-May-2014].

[14] NVIDIA. cuBLAS Library User Guide. `http://docs.nvidia.com/cuda/cublas/index.html`, February 2014. [Online; accessed 25-May-2014].

[15] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, February 2014. [Online; accessed 23-May-2014].

[16] NVIDIA. cuSPARSE Library User Guide. `http://docs.nvidia.com/cuda/cusparse/index.html`, February 2014. [Online; accessed 25-May-2014].

[17] Khronos Opencl and A. Munshi. The OpenCL Specification. `http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`, February 2014. [Online; accessed 23-May-2014].

[18] Ryan Smith. Nvidia geforce gtx 680 review: Retaking the performance crown. `http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review`, March 2012. [Online; accessed 23-May-2014].

[19] D.G. Spampinato and M. Püschel. A Basic Linear Algebra Compiler. In *International Symposium on Code Generation and Optimization (CGO)*, 2014.

[20] P. E. Vincent, P. Castonguay, and A. Jameson. A New Class of High-Order Energy Stable Flux Reconstruction Schemes. *J. Sci. Comput.*, 47(1):50–72, April 2011.

[21] R.C Whaley and J.J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[22] D.M. Williams and A. Jameson. Energy stable flux reconstruction schemes for advection-diffusion problems on tetrahedra. *Journal of Scientific Computing*, pages 1—-39, 2013.

[23] F.D. Witherden, A.M Farrington, and P.E. Vincent. PyFR: An Open Source Framework for Solving Advection-Diffusion Type Problems on Streaming Architectures using the Flux Reconstruction Approach. *arXiv preprint arXiv:1312.1638*, 2013.

[24] C. Woolley. Introduction to OpenCL. `http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf`, April 2011. [Online; accessed 14-May-2014].

Appendix A

Characteristics of the Operator Matrices

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	8	4	0.50
$M132$	4	8	0.50
$M3$	4	8	0.50
$M460$	8	4	0.50
$M6$	8	8	0.75

(a) Order = 1

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	12	9	0.67
$M132$	9	18	0.70
$M3$	9	12	0.67
$M460$	18	9	0.70
$M6$	18	12	0.83

(b) Order = 2

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	16	16	0.75
$M132$	16	32	0.75
$M3$	16	16	0.75
$M460$	32	16	0.75
$M6$	32	16	0.88

(c) Order = 3

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	20	25	0.80
$M132$	25	50	0.81
$M3$	25	20	0.80
$M460$	50	25	0.81
$M6$	50	20	0.90

(d) Order = 4

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	24	36	0.83
$M132$	36	72	0.83
$M3$	36	24	0.83
$M460$	72	36	0.83
$M6$	72	24	0.92

(e) Order = 5

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	28	49	0.86
$M132$	49	98	0.86
$M3$	49	28	0.86
$M460$	98	49	0.86
$M6$	98	28	0.93

(f) Order = 6

Table A.1: Height, width and sparsity factor of quadraliteral matrices obtained using the Gauss-Legendre method.

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	12	9	0.89
$M132$	9	18	0.70
$M3$	9	12	0.67
$M460$	18	9	0.70
$M6$	18	12	0.83

(g) Order = 2

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	16	16	0.94
$M132$	16	32	0.78
$M3$	16	16	0.75
$M460$	32	16	0.78
$M6$	32	16	0.88

(h) Order = 3

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	20	25	0.96
$M132$	25	50	0.82
$M3$	25	20	0.80
$M460$	50	25	0.82
$M6$	50	20	0.90

(i) Order = 4

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	24	36	0.97
$M132$	36	72	0.85
$M3$	36	24	0.83
$M460$	72	36	0.85
$M6$	72	24	0.92

(j) Order = 5

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	28	49	0.98
$M132$	49	98	0.87
$M3$	49	28	0.86
$M460$	98	49	0.87
$M6$	98	28	0.93

(k) Order = 6

Table A.1: Height, width and sparsity factor of quadraliteral matrices obtained using the Gauss-Legendre-Lobatto method.

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	24	8	0.75
$M132$	8	24	0.75
$M3$	8	24	0.75
$M460$	24	8	0.75
$M6$	24	24	0.92

(a) Order = 1

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	54	27	0.89
$M132$	27	81	0.90
$M3$	27	54	0.89
$M460$	81	27	0.90
$M6$	81	54	0.96

(b) Order = 2

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	96	64	0.94
$M132$	64	192	0.94
$M3$	64	96	0.94
$M460$	192	64	0.94
$M6$	192	96	0.98

(c) Order = 3

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	150	125	0.96
$M132$	125	375	0.96
$M3$	125	150	0.96
$M460$	375	125	0.96
$M6$	375	150	0.99

(d) Order = 4

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	216	216	0.97
$M132$	216	648	0.97
$M3$	216	216	0.97
$M460$	648	216	0.97
$M6$	648	216	0.99

(e) Order = 5

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	294	343	0.98
$M132$	343	1029	0.98
$M3$	343	294	0.98
$M460$	1029	343	0.98
$M6$	1029	294	0.99

(f) Order = 6

Table A.2: Height, width and sparsity factor of hexahedral matrices obtained using the Gauss-Legendre method.

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	54	27	0.96
$M132$	27	81	0.90
$M3$	27	54	0.89
$M460$	81	27	0.90
$M6$	81	54	0.96

(g) Order = 2

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	96	64	0.98
$M132$	64	192	0.95
$M3$	64	96	0.94
$M460$	192	64	0.95
$M6$	192	96	0.98

(h) Order = 3

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	150	125	0.99
$M132$	125	375	0.96
$M3$	125	150	0.96
$M460$	375	125	0.96
$M6$	375	150	0.99

(i) Order = 4

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	216	216	0.99
$M132$	216	648	0.98
$M3$	216	216	0.97
$M460$	648	216	0.98
$M6$	648	216	0.99

(j) Order = 5

Matrix	m	k	$\frac{zeros}{m\times k}$
$M0$	294	343	0.99
$M132$	343	1029	0.98
$M3$	343	294	0.98
$M460$	1029	343	0.98
$M6$	1029	294	0.99

(k) Order = 6

Table A.2: Height, width and sparsity factor of hexahedral matrices obtained using the Gauss-Legendre-Lobatto method.

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	6	3	0.00
$M132$	3	6	0.33
$M3$	3	6	0.00
$M460$	6	3	0.33
$M6$	6	6	0.33

(a) Order = 1

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	9	6	0.00
$M132$	6	12	0.11
$M3$	6	9	0.00
$M460$	12	6	0.11
$M6$	12	9	0.33

(b) Order = 2

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	12	10	0.00
$M132$	10	20	0.04
$M3$	10	12	0.00
$M460$	20	10	0.04
$M6$	20	12	0.33

(c) Order = 3

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	15	15	0.00
$M132$	15	30	0.04
$M3$	15	15	0.00
$M460$	30	15	0.04
$M6$	30	15	0.33

(d) Order = 4

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	18	21	0.00
$M132$	21	42	0.02
$M3$	21	18	0.00
$M460$	42	21	0.02
$M6$	42	18	0.33

(e) Order = 5

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	21	28	0.00
$M132$	28	56	0.02
$M3$	28	21	0.00
$M460$	56	28	0.02
$M6$	56	21	0.33

(f) Order = 6

Table A.3: Height, width and sparsity factor of triangular matrices obtained using the Williams-Shunn method.

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	12	4	0.00
$M132$	4	12	0.50
$M3$	4	12	0.00
$M460$	12	4	0.50
$M6$	12	12	0.50

(a) Order = 1

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	24	10	0.00
$M132$	10	30	0.16
$M3$	10	24	0.00
$M460$	30	10	0.16
$M6$	30	24	0.50

(b) Order = 2

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	40	20	0.00
$M132$	20	60	0.09
$M3$	20	40	0.00
$M460$	60	20	0.09
$M6$	60	40	0.50

(c) Order = 3

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	60	35	0.00
$M132$	35	105	0.07
$M3$	35	60	0.00
$M460$	105	35	0.07
$M6$	105	60	0.50

(d) Order = 4

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	84	56	0.00
$M132$	56	168	0.05
$M3$	56	84	0.00
$M460$	168	56	0.05
$M6$	168	84	0.50

(e) Order = 5

Matrix	m	k	$\frac{zeros}{m \times k}$
$M0$	112	84	0.00
$M132$	84	252	0.04
$M3$	84	112	0.00
$M460$	252	84	0.04
$M6$	252	112	0.50

(f) Order = 6

Table A.4: Height, width and sparsity factor of tetrahedral matrices obtained using the Shunn-Ham method.

Appendix B

Results for Individual Optimisations

The tables in this appendix give the raw experimental data obtained through benchmarking a set of kernels as described in Section 4.2. The results for kernels with the optimisations switched off are identical to GiMMiK's kernels, and are not duplicated in the following tables. See Appendix D for the benchmarking results for GiMMiK.

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.033	0.027	0.031	0.069	0.057	0.049	0.055	0.069
	M132	0.035	0.031	0.035	0.061	0.047	0.041	0.047	0.061
	M3	0.036	0.031	0.035	0.061	0.047	0.041	0.047	0.061
	M460	0.031	0.027	0.031	0.069	0.057	0.049	0.055	0.069
	M6	0.043	0.037	0.042	0.115	0.067	0.059	0.068	0.115
Order = 2	M0	0.059	0.049	0.055	0.188	0.094	0.081	0.093	0.188
	M132	0.080	0.068	0.078	0.289	0.104	0.088	0.103	0.289
	M3	0.060	0.051	0.058	0.204	0.082	0.074	0.086	0.204
	M460	0.072	0.062	0.071	0.295	0.126	0.107	0.124	0.295
	M6	0.077	0.070	0.079	0.376	0.130	0.114	0.133	0.376
Order = 3	M0	0.123	0.075	0.085	0.409	0.174	0.117	0.134	0.409
	M132	0.220	0.117	0.140	0.746	0.229	0.155	0.197	0.747
	M3	0.094	0.075	0.086	0.409	0.134	0.115	0.134	0.409
	M460	0.232	0.114	0.132	0.809	0.302	0.190	0.220	0.810
	M6	0.126	0.111	0.132	0.809	0.222	0.189	0.219	0.809
Order = 4	M0	0.299	0.108	0.126	0.739	0.234	0.160	0.190	0.740
	M132	0.683	0.186	0.230	1.845	0.550	0.265	0.418	1.846
	M3	0.171	0.106	0.122	0.792	0.233	0.166	0.201	0.792
	M460	0.874	0.182	0.211	1.868	0.512	0.315	0.365	1.867
	M6	0.183	0.162	0.194	1.538	0.334	0.283	0.344	1.537
Order = 5	M0	0.421	0.148	0.170	1.238	0.446	0.228	0.332	1.238
	M132	1.314	0.268	0.418	3.604	0.896	0.386	1.108	3.602
	M3	0.301	0.144	0.169	1.271	0.328	0.231	0.279	1.271
	M460	1.675	0.265	0.301	3.674	1.324	0.512	0.793	3.674
	M6	0.248	0.225	0.267	2.530	0.486	0.400	0.490	2.528
Order = 6	M0	0.674	0.195	0.223	1.935	0.599	0.283	0.446	1.936
	M132	2.460	0.382	0.712	6.781	1.578	0.678	1.849	6.779
	M3	0.410	0.186	0.223	2.031	0.432	0.307	0.397	2.032
	M460	2.811	0.368	0.431	6.779	2.033	0.701	1.233	6.783
	M6	0.350	0.299	0.353	3.994	0.648	0.531	0.715	3.997

(a) Quadrangles, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 2	M0	0.052	0.046	0.055	0.188	0.088	0.077	0.093	0.188
	M132	0.082	0.068	0.077	0.289	0.103	0.087	0.103	0.289
	M3	0.060	0.050	0.058	0.204	0.082	0.074	0.086	0.204
	M460	0.072	0.062	0.071	0.295	0.126	0.108	0.125	0.295
	M6	0.077	0.070	0.079	0.375	0.130	0.114	0.133	0.376
Order = 3	M0	0.074	0.065	0.084	0.409	0.119	0.105	0.133	0.409
	M132	0.201	0.119	0.141	0.746	0.221	0.156	0.239	0.747
	M3	0.092	0.075	0.086	0.409	0.133	0.115	0.133	0.409
	M460	0.200	0.113	0.132	0.809	0.286	0.190	0.219	0.809
	M6	0.126	0.110	0.132	0.809	0.221	0.189	0.219	0.810
Order = 4	M0	0.094	0.084	0.122	0.739	0.149	0.132	0.190	0.739
	M132	0.584	0.188	0.236	1.846	0.512	0.267	0.419	1.844
	M3	0.128	0.107	0.123	0.792	0.201	0.170	0.201	0.791
	M460	0.738	0.180	0.211	1.867	0.497	0.314	0.365	1.868
	M6	0.182	0.161	0.195	1.538	0.334	0.283	0.344	1.538
Order = 5	M0	0.116	0.101	0.171	1.238	0.181	0.160	0.334	1.238
	M132	1.171	0.278	0.455	3.602	0.853	0.486	1.104	3.603
	M3	0.307	0.144	0.169	1.272	0.329	0.231	0.279	1.271
	M460	1.485	0.265	0.334	3.674	1.236	0.510	0.793	3.675
	M6	0.251	0.225	0.267	2.529	0.483	0.399	0.489	2.529
Order = 6	M0	0.137	0.123	0.224	1.935	0.213	0.189	0.445	1.935
	M132	2.244	0.378	1.222	6.774	1.478	0.678	1.843	6.781
	M3	0.341	0.191	0.224	2.031	0.415	0.345	0.397	2.032
	M460	2.349	0.367	0.494	6.782	1.865	0.697	1.234	6.783
	M6	0.350	0.297	0.353	3.995	0.648	0.547	0.716	3.995

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Table B.1: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for quadrilateral element matrices in double precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.082	0.074	0.086	0.333	0.151	0.134	0.153	0.333
	M132	0.090	0.082	0.093	0.292	0.112	0.099	0.117	0.292
	M3	0.093	0.081	0.092	0.292	0.112	0.099	0.116	0.292
	M460	0.081	0.074	0.083	0.333	0.151	0.134	0.153	0.333
	M6	0.128	0.111	0.132	0.851	0.197	0.169	0.209	0.851
Order = 2	M0	0.485	0.200	0.226	2.161	0.697	0.376	0.397	2.158
	M132	0.477	0.286	0.438	3.095	0.535	0.410	0.955	3.095
	M3	0.328	0.201	0.320	2.076	0.441	0.285	0.463	2.076
	M460	0.595	0.263	0.306	3.212	0.983	0.532	0.556	3.211
	M6	0.361	0.324	0.408	6.166	0.684	0.522	1.102	6.163
Order = 3	M0	1.333	0.398	0.516	8.503	1.451	0.962	2.446	8.497
	M132	3.772	0.823	5.182	16.51	2.427	0.982	27.40	16.52
	M3	0.785	0.417	1.601	8.464	1.018	0.733	2.293	8.464
	M460	3.537	0.632	0.963	16.96	3.089	1.795	4.706	16.96
	M6	0.824	0.723	1.627	25.26	2.212	1.779	6.293	25.25
Order = 4	M0	5.095	0.723	4.080	25.64	3.465	1.617	7.279	25.66
	M132	14.04	4.141	29.73	62.89	6.958	3.846	207.12	62.86
	M3	2.269	0.691	6.697	25.39	2.118	1.337	16.06	25.41
	M460	9.518	1.496	6.231	63.77	7.692	3.682	17.25	63.77
	M6	2.390	1.321	4.687	75.53	4.251	3.404	50.41	75.45
Order = 5	M0	11.51	4.793		62.28	7.547	5.129		62.24
	M132	33.49	11.07		186.13	15.32	9.426		185.99
	M3	5.728	1.319		62.28	4.311	2.371		62.26
	M460	26.33	5.607		186.56	16.70	9.181		186.61
	M6	5.803	2.002		186.53	7.647	5.838		186.61
Order = 6	M0	20.81	10.37		134.34	13.43	10.37		134.23
	M132	69.47	22.79		468.79	30.61	17.65		468.80
	M3	10.57	3.652		134.43	7.264	4.644		134.49
	M460	52.36	11.72		469.18	29.72	18.05		468.92
	M6	10.81	3.019		402.11	13.32	9.221		402.26

(c) Hexahedra, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 2	M0	0.219	0.191	0.228	2.159	0.418	0.322	0.397	2.158
	M132	0.509	0.290	0.437	3.097	0.545	0.423	0.955	3.094
	M3	0.335	0.201	0.313	2.076	0.442	0.284	0.464	2.075
	M460	0.595	0.263	0.306	3.212	0.983	0.533	0.557	3.211
	M6	0.361	0.323	0.407	6.165	0.684	0.522	1.103	6.165
Order = 3	M0	0.425	0.376	0.520	8.499	1.039	0.665	2.450	8.495
	M132	2.817	0.773	6.097	16.51	2.161	0.970	28.27	16.51
	M3	0.817	0.420	1.623	8.462	1.022	0.734	2.288	8.464
	M460	2.612	0.632	1.830	16.96	2.666	1.777	4.696	16.96
	M6	0.807	0.716	1.639	25.27	2.183	1.310	6.290	25.26
Order = 4	M0	0.696	0.612	2.287	25.65	1.671	1.063	7.340	25.64
	M132	12.92	3.549	30.22	62.85	6.707	3.405	201.55	62.83
	M3	2.015	0.674	6.265	25.39	2.031	1.329	16.52	25.40
	M460	8.650	1.441	18.54	63.76	7.380	3.690	17.27	63.75
	M6	2.391	1.319	4.925	75.50	4.249	3.403	49.98	75.51
Order = 5	M0	1.060	0.915	12.60	62.32	2.462	1.996	131.26	62.28
	M132	29.71	10.51		186.04	14.54	8.430		186.12
	M3	6.023	1.345		62.26	4.242	2.366		62.32
	M460	21.23	4.959		186.60	15.13	8.639		186.54
	M6	5.148	1.970		186.53	7.431	5.759		186.54
Order = 6	M0	2.086	1.273		134.23	3.506	2.749		134.26
	M132	59.75	22.09		468.82	28.78	16.54		468.77
	M3	8.839	3.235		134.11	6.885	4.302		134.30
	M460	47.44	11.18		469.21	28.09	16.74		468.94
	M6	10.81	3.040		402.02	13.32	9.235		401.80

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table B.1: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for hexahedral element matrices in double precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.027	0.020	0.023	0.052	0.044	0.038	0.043	0.052
	M132	0.027	0.025	0.027	0.044	0.036	0.032	0.036	0.044
	M3	0.029	0.024	0.028	0.044	0.037	0.032	0.036	0.044
	M460		0.021	0.023	0.052	0.042	0.038	0.043	0.052
	M6	0.037	0.027	0.031	0.083	0.054	0.045	0.051	0.083
Order = 2	M0	0.050	0.034	0.039	0.116	0.072	0.060	0.066	0.116
	M132	0.061	0.046	0.053	0.144	0.073	0.062	0.069	0.144
	M3	0.052	0.037	0.042	0.114	0.066	0.054	0.060	0.114
	M460	0.059	0.042	0.047	0.135	0.088	0.072	0.081	0.135
	M6	0.069	0.049	0.055	0.188	0.096	0.080	0.091	0.188
Order = 3	M0	0.114	0.051	0.057	0.204	0.123	0.084	0.095	0.204
	M132	0.273	0.077	0.087	0.342	0.247	0.103	0.114	0.342
	M3	0.114	0.053	0.059	0.221	0.115	0.080	0.091	0.221
	M460	0.188	0.069	0.078	0.337	0.192	0.121	0.137	0.336
	M6	0.158	0.075	0.084	0.394	0.169	0.126	0.144	0.394
Order = 4	M0	0.328	0.071	0.081	0.374	0.264	0.112	0.127	0.374
	M132	1.181	0.117	0.135	0.680	0.792	0.189	0.186	0.680
	M3	0.312	0.071	0.081	0.374	0.255	0.114	0.126	0.374
	M460	0.751	0.109	0.123	0.742	0.611	0.182	0.206	0.742
	M6	0.330	0.109	0.120	0.742	0.284	0.185	0.205	0.742
Order = 5	M0	0.663	0.095	0.109	0.607	0.545	0.144	0.162	0.607
	M132	2.710	0.270	0.212	1.327	1.887	0.463	0.360	1.327
	M3	0.592	0.094	0.105	0.618	0.442	0.158	0.167	0.618
	M460	1.762	0.163	0.187	1.356	1.459	0.268	0.306	1.356
	M6	0.970	0.148	0.162	1.193	0.605	0.262	0.281	1.192
Order = 6	M0	1.635	0.124	0.143	0.908	1.332	0.181	0.207	0.907
	M132	5.514	0.444	0.440	2.195	3.399	0.705	0.860	2.194
	M3	1.040	0.122	0.136	0.885	0.739	0.250	0.224	0.885
	M460	6.123	0.275	0.324	2.267	4.589	0.408	0.475	2.265
	M6	1.555	0.194	0.213	1.759	0.931	0.345	0.384	1.759

(e) Triangles, Williams-Shunn

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.044	0.037	0.042	0.101	0.078	0.067	0.075	0.101
	M132	0.047	0.041	0.048	0.085	0.057	0.052	0.058	0.085
	M3	0.052	0.043	0.048	0.085	0.058	0.052	0.059	0.085
	M460		0.037	0.042	0.101	0.077	0.067	0.075	0.101
	M6	0.075	0.056	0.063	0.239	0.102	0.088	0.101	0.239
Order = 2	M0	0.174	0.082	0.090	0.402	0.191	0.145	0.159	0.402
	M132	0.202	0.102	0.118	0.494	0.212	0.140	0.152	0.494
	M3	0.190	0.087	0.099	0.402	0.196	0.126	0.129	0.402
	M460	0.214	0.094	0.104	0.518	0.231	0.168	0.190	0.518
	M6	0.503	0.130	0.148	1.099	0.378	0.241	0.244	1.100
Order = 3	M0	1.420	0.149	0.169	1.207	0.973	0.322	0.289	1.208
	M132	2.543	0.249	0.313	1.695	1.658	0.446	0.617	1.694
	M3	1.278	0.169	0.183	1.145	0.913	0.349	0.315	1.146
	M460	2.079	0.204	0.235	1.804	1.390	0.370	0.404	1.805
	M6	3.171	0.264	0.294	3.389	1.704	0.717	0.716	3.390
Order = 4	M0	6.345	0.308	0.306	2.991	5.020	0.866	0.669	2.991
	M132	12.10	2.188	1.257	5.132	8.917	2.978	1.652	5.132
	M3	4.915	0.490	0.433	2.973	2.967	0.719	0.958	2.973
	M460	12.76	0.792	0.602	5.273	9.268	0.982	1.207	5.273
	M6	8.614	0.458	1.065	8.838	4.707	1.359	2.558	8.838
Order = 5	M0	16.00	1.294	1.479	6.532	9.795	1.610	2.018	6.533
	M132	44.69	9.915	6.756	12.68	38.12	14.28	12.78	12.68
	M3	13.47	2.670	1.541	6.499	8.035	3.133	1.977	6.503
	M460	34.43	2.370	2.915	13.02	20.06	4.100	5.186	13.02
	M6	21.97	1.747	2.382	19.43	9.748	2.654	5.395	19.39
Order = 6	M0	35.52	2.905	2.785	12.94	21.53	4.322	3.665	12.95
	M132	120.51	27.30	26.81	28.27	115.74	45.17	50.56	28.28
	M3	32.12	6.254	2.880	12.89	23.07	7.535	3.538	12.88
	M460	85.79	6.790	7.236	29.07	52.05	9.066	11.19	29.08
	M6	42.25	4.012	4.623	38.52	22.77	5.308	10.04	38.53

(f) Tetrahedra, Shunn-Ham

Table B.1: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for triangular and tetrahedral element matrices in double precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0		0.016	0.017	0.050		0.032	0.032	0.050
	M132		0.020	0.021	0.044		0.027	0.027	0.044
	M3		0.020	0.021	0.044		0.027	0.027	0.044
	M460		0.016	0.017	0.050		0.032	0.033	0.050
	M6		0.024	0.023	0.083		0.037	0.038	0.083
Order = 2	M0		0.028	0.028	0.135		0.050	0.050	0.135
	M132		0.041	0.041	0.210		0.057	0.056	0.210
	M3		0.031	0.031	0.147		0.045	0.046	0.147
	M460		0.036	0.036	0.210		0.065	0.066	0.210
	M6		0.040	0.040	0.259		0.069	0.070	0.259
Order = 3	M0		0.043	0.043	0.280		0.071	0.075	0.280
	M132		0.073	0.072	0.530		0.096	0.099	0.530
	M3		0.043	0.043	0.280		0.074	0.072	0.280
	M460		0.064	0.064	0.551		0.112	0.115	0.552
	M6		0.064	0.064	0.552		0.119	0.114	0.551
Order = 4	M0		0.064	0.064	0.528		0.100	0.101	0.527
	M132		0.110	0.119	1.215		0.167	0.165	1.211
	M3		0.061	0.061	0.563		0.104	0.107	0.563
	M460		0.102	0.105	1.327		0.187	0.190	1.326
	M6		0.097	0.095	1.071		0.177	0.181	1.071
Order = 5	M0		0.087	0.088	0.849		0.133	0.137	0.849
	M132		0.157	0.209	2.310		0.300	0.347	2.306
	M3		0.082	0.082	0.897		0.143	0.147	0.897
	M460		0.147	0.150	2.517		0.287	0.271	2.518
	M6		0.127	0.129	1.784		0.246	0.249	1.784
Order = 6	M0		0.112	0.115	1.278		0.164	0.185	1.278
	M132		0.220	0.330	4.337		0.659	0.524	4.361
	M3		0.107	0.109	1.456		0.187	0.193	1.454
	M460		0.200	0.222	4.448		0.394	0.457	4.448
	M6		0.171	0.169	2.833		0.328	0.332	2.831

(a) Quadrangles, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 2	M0		0.026	0.028	0.135		0.048	0.050	0.135
	M132		0.041	0.041	0.210		0.055	0.058	0.210
	M3		0.031	0.030	0.147		0.046	0.046	0.147
	M460		0.036	0.036	0.210		0.065	0.066	0.210
	M6		0.040	0.040	0.259		0.069	0.070	0.259
Order = 3	M0		0.037	0.043	0.280		0.063	0.071	0.280
	M132		0.072	0.072	0.530		0.097	0.099	0.530
	M3		0.043	0.043	0.280		0.074	0.071	0.280
	M460		0.064	0.065	0.551		0.120	0.123	0.552
	M6		0.066	0.064	0.552		0.112	0.114	0.551
Order = 4	M0		0.049	0.063	0.527		0.079	0.098	0.527
	M132		0.110	0.122	1.210		0.165	0.168	1.209
	M3		0.061	0.061	0.563		0.105	0.108	0.563
	M460		0.103	0.104	1.327		0.185	0.191	1.327
	M6		0.097	0.095	1.071		0.177	0.181	1.071
Order = 5	M0		0.059	0.087	0.849		0.094	0.126	0.849
	M132		0.157	0.229	2.307		0.296	0.353	2.306
	M3		0.083	0.082	0.897		0.142	0.148	0.897
	M460		0.148	0.175	2.515		0.287	0.336	2.517
	M6		0.132	0.129	1.784		0.246	0.248	1.784
Order = 6	M0		0.071	0.114	1.278		0.117	0.165	1.277
	M132		0.213	0.630	4.335		0.649	0.692	4.346
	M3		0.108	0.108	1.455		0.186	0.193	1.456
	M460		0.208	0.251	4.447		0.393	0.634	4.453
	M6		0.171	0.169	2.833		0.329	0.332	2.831

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Table B.2: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for quadrilateral element matrices in single precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Order	Matrix	$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0		0.043	0.043	0.236		0.080	0.081	0.236
	M132		0.050	0.048	0.208		0.063	0.063	0.208
	M3		0.050	0.049	0.208		0.062	0.063	0.208
	M460		0.043	0.043	0.236		0.080	0.080	0.236
	M6		0.065	0.066	0.601		0.106	0.109	0.601
Order = 2	M0		0.115	0.114	1.535		0.199	0.220	1.537
	M132		0.161	0.210	1.970		0.360	0.295	1.969
	M3		0.117	0.145	1.352		0.209	0.194	1.349
	M460		0.147	0.153	2.298		0.280	0.309	2.299
	M6		0.189	0.205	4.029		0.353	0.500	4.021
Order = 3	M0		0.227	0.265	5.468		0.569	0.625	5.475
	M132		0.416	2.369	10.47		0.889	2.374	10.47
	M3		0.231	1.448	5.344		0.684	1.492	5.345
	M460		0.358	0.482	10.91		1.045	1.153	10.91
	M6		0.405	1.305	15.96		1.755	2.090	15.96
Order = 4	M0		0.392	2.025	16.29		1.529	2.656	16.28
	M132		2.247	11.54	39.94		3.023	13.56	39.94
	M3		0.393	4.160	16.15		1.279	4.271	16.14
	M460		0.725	3.702	40.45		3.586	4.898	40.46
	M6		0.757	2.917	48.00		3.391	4.315	47.99
Order = 5	M0		0.636	4.768	39.42		2.605		39.42
	M132		7.682	37.37	119.33		7.066		119.34
	M3		0.815	9.993	39.42		2.392		39.44
	M460		1.963	5.809	118.15		6.928		118.21
	M6		1.427	5.789	118.14		6.091		118.16
Order = 6	M0		4.220		85.50		5.654		85.51
	M132		14.97		301.28		14.47		301.34
	M3		1.813		85.39		4.150		85.42
	M460		5.597		298.37		12.05		298.25
	M6		2.129		255.74		9.105		255.82

(c) Hexahedra, Gauss-Legendre Method

Order	Matrix	$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 2	M0		0.111	0.112	1.537		0.195	0.202	1.535
	M132		0.158	0.220	1.969		0.355	0.292	1.968
	M3		0.118	0.146	1.352		0.211	0.194	1.350
	M460		0.149	0.153	2.297		0.301	0.309	2.298
	M6		0.188	0.206	4.023		0.353	0.500	4.021
Order = 3	M0		0.215	0.255	5.468		0.526	0.588	5.464
	M132		0.398	2.806	10.47		0.850	2.806	10.47
	M3		0.232	1.451	5.352		0.692	1.488	5.343
	M460		0.357	1.728	10.91		1.038	1.288	10.92
	M6		0.416	1.307	15.96		1.718	2.058	15.96
Order = 4	M0		0.347	1.930	16.29		0.638	2.393	16.29
	M132		2.264	13.35	39.93		2.463	12.52	39.94
	M3		0.398	4.166	16.15		1.249	4.273	16.15
	M460		0.727	5.940	40.45		3.554	6.608	40.45
	M6		0.757	2.887	48.02		3.393	4.340	48.01
Order = 5	M0		0.519	4.721	39.42		1.162	5.266	39.44
	M132		6.023	45.13	119.33		6.043		119.23
	M3		0.746	9.988	39.42		2.396		39.42
	M460		1.852	17.43	118.21		6.773		118.16
	M6		1.302	5.846	118.11		5.627		118.12
Order = 6	M0		1.067	14.35	85.53		2.891		85.52
	M132		15.56		301.23		13.86		301.36
	M3		1.761		85.39		3.721		85.37
	M460		5.038		298.26		11.47		298.26
	M6		2.067		255.77		9.102		255.84

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table B.2: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for hexahedral element matrices in single precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0		0.014	0.014	0.037		0.024	0.025	0.037
	M132		0.016	0.016	0.032		0.020	0.021	0.032
	M3		0.016	0.017	0.032		0.020	0.020	0.032
	M460		0.013	0.013	0.037		0.024	0.024	0.037
	M6		0.017	0.019	0.059		0.030	0.030	0.059
Order = 2	M0		0.020	0.020	0.084		0.038	0.038	0.084
	M132		0.028	0.028	0.100		0.038	0.038	0.100
	M3		0.023	0.024	0.081		0.034	0.034	0.081
	M460		0.024	0.024	0.098		0.046	0.046	0.097
	M6		0.027	0.028	0.135		0.050	0.050	0.135
Order = 3	M0		0.029	0.029	0.143		0.052	0.052	0.143
	M132		0.047	0.046	0.241		0.066	0.066	0.241
	M3		0.031	0.032	0.153		0.050	0.050	0.153
	M460		0.040	0.040	0.235		0.074	0.074	0.235
	M6		0.043	0.043	0.272		0.081	0.077	0.272
Order = 4	M0		0.040	0.041	0.261		0.069	0.069	0.261
	M132		0.069	0.070	0.485		0.099	0.109	0.485
	M3		0.041	0.042	0.260		0.073	0.073	0.260
	M460		0.061	0.062	0.513		0.110	0.111	0.512
	M6		0.061	0.061	0.512		0.117	0.109	0.512
Order = 5	M0		0.057	0.057	0.429		0.095	0.095	0.429
	M132		0.105	0.108	0.892		0.185	0.187	0.890
	M3		0.054	0.053	0.436		0.097	0.097	0.436
	M460		0.087	0.088	0.956		0.175	0.175	0.956
	M6		0.083	0.084	0.823		0.158	0.161	0.823
Order = 6	M0		0.074	0.075	0.658		0.118	0.118	0.658
	M132		0.174	0.162	1.431		0.371	0.372	1.428
	M3		0.068	0.068	0.624		0.126	0.126	0.624
	M460		0.132	0.132	1.606		0.237	0.238	1.604
	M6		0.107	0.105	1.238		0.206	0.212	1.238

(e) Triangles, Williams-Shunn

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0		0.022	0.022	0.073		0.043	0.043	0.073
	M132		0.026	0.026	0.061		0.032	0.032	0.061
	M3		0.026	0.026	0.061		0.033	0.033	0.061
	M460		0.022	0.022	0.073		0.043	0.043	0.073
	M6		0.032	0.031	0.166		0.054	0.054	0.166
Order = 2	M0		0.046	0.046	0.280		0.091	0.091	0.280
	M132		0.062	0.062	0.351		0.084	0.084	0.351
	M3		0.052	0.052	0.285		0.071	0.070	0.285
	M460		0.053	0.054	0.359		0.101	0.101	0.359
	M6		0.074	0.074	0.777		0.138	0.129	0.777
Order = 3	M0		0.085	0.083	0.841		0.182	0.182	0.841
	M132		0.180	0.161	1.098		0.410	0.419	1.098
	M3		0.094	0.096	0.766		0.325	0.325	0.766
	M460		0.110	0.114	1.256		0.222	0.223	1.255
	M6		0.150	0.150	2.259		0.416	0.315	2.257
Order = 4	M0		0.165	0.168	2.068		0.507	0.507	2.066
	M132		0.433	0.599	3.306		0.941	0.957	3.302
	M3		0.239	0.207	1.920		0.653	0.653	1.921
	M460		0.270	0.277	3.662		0.502	0.504	3.662
	M6		0.268	0.504	5.707		0.798	0.824	5.722
Order = 5	M0		1.084	0.759	4.236		1.625	1.625	4.227
	M132		2.602	2.660	8.056		2.827	2.875	8.054
	M3		0.817	0.797	4.130		1.280	1.278	4.129
	M460		1.307	1.412	8.428		1.847	1.862	8.444
	M6		1.632	1.236	12.32		2.595	2.664	12.31
Order = 6	M0		2.340	1.459	8.226		2.879	2.882	8.224
	M132		9.440	6.062	17.96		10.52	10.47	17.97
	M3		2.239	2.640	8.202		2.621	2.624	8.207
	M460		6.940	5.695	18.46		6.889	6.901	18.46
	M6		3.203	4.136	24.53		4.598	4.656	24.52

(f) Tetrahedra, Shunn-Ham

Table B.2: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for triangular and tetrahedral element matrices in single precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.027	0.024	0.028	0.069	0.038	0.036	0.038	0.060
	M132	0.024	0.024	0.027	0.063	0.031	0.031	0.036	0.063
	M3	0.024	0.025	0.039	0.063	0.032	0.032	0.036	0.063
	M460	0.021	0.021	0.023	0.069	0.037	0.036	0.037	0.068
	M6	0.029	0.027	0.035	0.122	0.045	0.044	0.055	0.122
Order = 2	M0	0.041	0.036	0.047	0.202	0.062	0.062	0.079	0.202
	M132	0.057	0.052	0.080	0.295	0.069	0.067	0.107	0.294
	M3	0.042	0.040	0.059	0.206	0.056	0.057	0.078	0.205
	M460	0.050	0.046	0.064	0.303	0.085	0.082	0.110	0.302
	M6	0.052	0.052	0.071	0.384	0.085	0.086	0.137	0.383
Order = 3	M0	0.089	0.058	0.086	0.443	0.124	0.088	0.158	0.443
	M132	0.164	0.093	0.217	0.773	0.162	0.124	0.316	0.854
	M3	0.067	0.061	0.119	0.401	0.090	0.085	0.158	0.443
	M460	0.174	0.086	0.141	0.796	0.218	0.142	0.306	0.878
	M6	0.088	0.082	0.103	0.796	0.143	0.137	0.316	0.878
Order = 4	M0	0.207	0.086	0.162	0.844	0.169	0.121	0.297	0.740
	M132	0.527	0.160	0.523	1.829	0.420	0.225	0.748	1.827
	M3	0.140	0.081	0.230	0.755	0.169	0.123	0.293	0.754
	M460	0.644	0.148	0.308	1.842	0.364	0.244	0.710	2.096
	M6	0.124	0.120	0.170	1.483	0.213	0.207	0.568	1.683
Order = 5	M0	0.333	0.115	0.255	1.257	0.332	0.191	0.495	1.294
	M132	0.974	0.282	0.944	3.693	0.696	0.357	1.594	3.697
	M3	0.215	0.112	0.385	1.445	0.235	0.174	0.480	1.268
	M460	1.285	0.255	0.440	3.869	0.981	0.447	1.474	3.756
	M6	0.174	0.174	0.213	2.531	0.314	0.285	0.971	2.530
Order = 6	M0	0.449	0.153	0.404	1.990	0.443	0.234	0.722	1.989
	M132	1.624	0.466	1.798	6.925	1.139	0.696	2.645	6.779
	M3	0.325	0.143	0.567	2.010	0.312	0.226	0.684	2.288
	M460	2.049	0.419	0.887	6.938	1.512	0.634	2.424	6.934
	M6	0.218	0.227	0.300	4.102	0.516	0.382	1.529	3.989

(a) Quadrangles, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 2	M0	0.034	0.034	0.040	0.202	0.056	0.057	0.080	0.202
	M132	0.056	0.050	0.079	0.294	0.069	0.067	0.107	0.266
	M3	0.042	0.039	0.060	0.205	0.056	0.057	0.078	0.186
	M460	0.051	0.046	0.066	0.303	0.083	0.082	0.111	0.274
	M6	0.052	0.054	0.060	0.384	0.086	0.085	0.138	0.347
Order = 3	M0	0.049	0.048	0.073	0.443	0.076	0.078	0.159	0.443
	M132	0.147	0.093	0.259	0.772	0.157	0.128	0.336	0.853
	M3	0.065	0.060	0.121	0.401	0.089	0.087	0.160	0.443
	M460	0.143	0.088	0.186	0.797	0.206	0.140	0.305	0.878
	M6	0.085	0.084	0.109	0.796	0.142	0.137	0.306	0.878
Order = 4	M0	0.061	0.061	0.141	0.843	0.097	0.096	0.297	0.762
	M132	0.469	0.156	0.593	1.882	0.399	0.222	0.737	1.824
	M3	0.100	0.084	0.220	0.756	0.133	0.126	0.294	0.755
	M460	0.553	0.139	0.445	1.843	0.356	0.239	0.710	1.842
	M6	0.124	0.118	0.163	1.484	0.213	0.203	0.564	1.684
Order = 5	M0	0.077	0.077	0.240	1.296	0.117	0.116	0.531	1.293
	M132	0.904	0.298	1.185	3.699	0.648	0.457	1.668	3.698
	M3	0.211	0.111	0.369	1.269	0.238	0.172	0.468	1.268
	M460	1.088	0.225	0.870	3.828	0.924	0.432	1.477	3.754
	M6	0.173	0.170	0.214	2.532	0.296	0.289	0.930	2.879
Order = 6	M0	0.091	0.090	0.394	1.991	0.137	0.138	0.811	1.988
	M132	1.628	0.434	2.123	6.786	1.066	0.660	2.839	6.780
	M3	0.208	0.138	0.607	2.011	0.299	0.272	0.740	2.285
	M460	1.827	0.386	1.524	6.940	1.399	0.615	2.517	6.935
	M6	0.218	0.215	0.307	3.991	0.518	0.391	1.347	3.991

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Table B.3: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for quadrilateral element matrices in double precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.057	0.054	0.067	0.355	0.102	0.099	0.131	0.355
	M132	0.059	0.063	0.096	0.328	0.074	0.073	0.125	0.293
	M3	0.062	0.062	0.121	0.301	0.074	0.075	0.125	0.292
	M460	0.055	0.054	0.064	0.326	0.099	0.099	0.131	0.317
	M6	0.085	0.082	0.164	0.888	0.126	0.124	0.341	0.865
Order = 2	M0	0.354	0.153	0.372	2.144	0.513	0.300	0.747	2.140
	M132	0.353	0.242	1.012	3.134	0.405	0.340	1.294	3.131
	M3	0.265	0.151	0.746	2.098	0.335	0.216	0.804	2.127
	M460	0.445	0.195	0.583	3.220	0.718	0.417	1.102	3.215
	M6	0.250	0.240	0.761	6.317	0.462	0.373	2.301	6.308
Order = 3	M0	0.896	0.302	1.187	9.355	0.945	0.816	3.442	8.770
	M132	2.348	0.709	5.379	17.15	1.523	0.900	19.55	17.13
	M3	0.628	0.351	2.587	8.658	0.699	0.623	3.739	8.617
	M460	2.546	0.479	1.510	17.54	2.006	1.552	6.868	17.54
	M6	0.573	0.520	2.056	25.80	1.581	1.437	9.896	25.77
Order = 4	M0	3.614	0.681	3.725	26.32	2.231	1.260	10.47	26.30
	M132	8.575	3.310	23.93	65.18	4.471	2.804	137.76	65.17
	M3	1.566	0.520	8.563	26.25	1.425	1.149	14.58	26.26
	M460	6.483	1.125	7.068	65.74	5.148	3.045	25.83	65.70
	M6	1.640	0.838	5.204	78.55	2.908	2.530	44.05	78.49
Order = 5	M0	7.463	3.602		65.00	5.001	3.848		64.91
	M132	21.27	8.280		194.54	9.260	6.942		193.64
	M3	4.158	1.195		64.99	3.109	2.058		64.89
	M460	16.94	4.421		194.93	10.86	7.132		194.72
	M6	3.840	1.454		194.80	4.888	4.635		194.66
Order = 6	M0	13.17	7.398		140.23	8.586	7.547		140.21
	M132	44.40	16.47		490.63	18.82	12.93		490.88
	M3	6.520	2.755		140.07	4.832	3.854		139.94
	M460	33.67	8.814		489.61	19.28	13.99		489.81
	M6	6.412	2.210		420.03	8.436	6.830		420.30

(c) Hexahedra, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 2	M0	0.145	0.144	0.231	2.143	0.279	0.232	0.821	2.272
	M132	0.370	0.239	1.008	3.159	0.405	0.348	1.422	3.133
	M3	0.263	0.151	0.747	2.102	0.335	0.216	0.897	2.098
	M460	0.445	0.201	0.586	3.221	0.718	0.417	1.205	3.277
	M6	0.247	0.239	0.762	6.320	0.460	0.373	2.558	6.313
Order = 3	M0	0.280	0.274	1.112	8.788	0.716	0.492	3.551	8.783
	M132	1.937	0.640	6.205	17.15	1.559	0.873	20.05	17.15
	M3	0.661	0.350	2.595	8.624	0.783	0.625	3.369	8.622
	M460	1.870	0.448	2.904	17.56	1.946	1.510	6.864	17.56
	M6	0.576	0.522	2.060	25.81	1.387	0.981	9.957	25.78
Order = 4	M0	0.405	0.440	3.259	26.33	1.091	0.780	10.53	26.32
	M132	7.708	2.854	25.07	65.24	4.309	2.773	138.89	65.17
	M3	1.377	0.493	8.472	26.26	1.379	1.115	14.95	26.24
	M460	5.662	1.052	15.23	65.73	4.974	2.960	25.82	65.69
	M6	1.639	0.848	5.385	78.52	3.017	2.524	44.08	78.48
Order = 5	M0	0.676	0.657	11.14	64.97	1.518	1.393	85.47	64.91
	M132	18.71	7.619		194.73	8.817	6.302		194.36
	M3	4.015	0.980		65.00	3.042	1.826		64.88
	M460	14.99	3.993		194.88	9.903	6.723		194.63
	M6	3.488	1.443		194.86	4.938	4.385		194.56
Order = 6	M0	1.349	0.911		140.18	2.272	1.944		140.08
	M132	37.85	16.12		490.28	17.61	12.15		490.74
	M3	5.754	2.446		140.08	4.793	3.296		139.93
	M460	30.83	8.355		489.37	18.19	12.81		489.67
	M6	6.479	2.227		419.58	8.414	6.824		420.31

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table B.3: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for hexahedral element matrices in double precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.021	0.014	0.027	0.042	0.030	0.025	0.028	0.042
	M132	0.019	0.017	0.019	0.034	0.025	0.021	0.025	0.034
	M3	0.022	0.018	0.020	0.038	0.027	0.022	0.025	0.038
	M460		0.014	0.016	0.047	0.028	0.025	0.028	0.047
	M6	0.028	0.022	0.028	0.076	0.036	0.030	0.037	0.075
Order = 2	M0	0.043	0.036	0.035	0.113	0.053	0.045	0.050	0.113
	M132	0.050	0.043	0.049	0.134	0.062	0.054	0.059	0.134
	M3	0.046	0.037	0.038	0.107	0.052	0.047	0.049	0.107
	M460	0.049	0.043	0.045	0.141	0.065	0.060	0.063	0.140
	M6	0.055	0.044	0.062	0.202	0.068	0.063	0.079	0.202
Order = 3	M0	0.097	0.069	0.070	0.218	0.102	0.089	0.089	0.218
	M132	0.221	0.109	0.115	0.345	0.196	0.127	0.130	0.346
	M3	0.100	0.070	0.072	0.218	0.099	0.087	0.089	0.217
	M460	0.158	0.106	0.108	0.360	0.156	0.134	0.136	0.360
	M6	0.121	0.085	0.099	0.423	0.132	0.113	0.152	0.423
Order = 4	M0	0.272	0.127	0.127	0.394	0.219	0.146	0.145	0.393
	M132	0.928	0.237	0.273	0.677	0.629	0.274	0.277	0.657
	M3	0.251	0.116	0.127	0.357	0.213	0.139	0.146	0.346
	M460	0.601	0.233	0.247	0.712	0.504	0.273	0.280	0.691
	M6	0.275	0.155	0.183	0.712	0.243	0.193	0.272	0.691
Order = 5	M0	0.544	0.205	0.207	0.649	0.453	0.229	0.241	0.648
	M132	2.006	0.541	0.521	1.338	1.394	0.650	0.591	1.335
	M3	0.419	0.199	0.205	0.597	0.361	0.226	0.234	0.596
	M460	1.254	0.504	0.526	1.358	1.104	0.544	0.561	1.356
	M6	0.674	0.261	0.311	1.163	0.498	0.357	0.450	1.129
Order = 6	M0	1.142	0.325	0.352	0.900	1.054	0.357	0.357	0.898
	M132	3.613	0.953	0.984	2.257	2.519	1.081	1.154	2.258
	M3	0.801	0.312	0.317	0.881	0.538	0.374	0.352	0.880
	M460	4.222	0.931	0.943	2.288	3.069	0.948	0.982	2.285
	M6	1.024	0.419	0.438	1.990	0.659	0.526	0.655	1.964

(e) Triangles, Williams-Shunn

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.034	0.030	0.029	0.089	0.055	0.052	0.051	0.089
	M132	0.028	0.032	0.035	0.078	0.041	0.040	0.046	0.078
	M3	0.038	0.036	0.037	0.089	0.048	0.046	0.046	0.089
	M460		0.027	0.027	0.101	0.051	0.049	0.051	0.101
	M6	0.051	0.045	0.057	0.256	0.071	0.067	0.098	0.257
Order = 2	M0	0.144	0.097	0.122	0.431	0.181	0.137	0.158	0.430
	M132	0.185	0.121	0.166	0.457	0.195	0.144	0.200	0.457
	M3	0.177	0.117	0.142	0.371	0.183	0.133	0.152	0.370
	M460	0.184	0.114	0.133	0.488	0.195	0.159	0.194	0.487
	M6	0.397	0.152	0.237	1.091	0.311	0.222	0.423	1.089
Order = 3	M0	1.078	0.413	0.431	1.222	0.746	0.485	0.471	1.348
	M132	1.892	0.549	0.707	1.730	1.257	0.627	0.831	1.781
	M3	0.956	0.364	0.464	1.167	0.796	0.454	0.499	1.167
	M460	1.412	0.532	0.651	2.020	1.049	0.677	0.701	1.776
	M6	2.154	0.531	0.717	3.475	1.446	0.813	1.444	3.782
Order = 4	M0	4.217	0.981	1.232	3.065	3.763	1.239	1.312	3.060
	M132	7.926	2.310	2.108	5.200	5.871	2.853	2.414	5.196
	M3	3.313	0.959	1.251	3.013	2.031	1.035	1.398	3.016
	M460	8.538	1.887	2.103	5.361	6.368	1.826	2.372	5.360
	M6	5.723	1.338	1.767	9.020	3.402	1.602	3.623	9.019
Order = 5	M0	10.79	2.268	2.453	6.738	6.776	2.410	2.714	6.737
	M132	28.95	7.861	5.936	13.13	24.50	10.92	9.105	13.15
	M3	9.083	2.493	2.438	6.660	5.574	3.129	2.702	6.656
	M460	23.03	4.802	5.072	13.45	13.54	5.426	6.139	13.46
	M6	14.69	2.952	3.866	19.90	7.071	3.630	8.050	19.92
Order = 6	M0	23.90	4.454	4.805	13.28	14.46	5.533	5.435	13.29
	M132	78.36	20.66	18.94	29.58	73.46	33.41	33.03	29.61
	M3	21.46	5.620	4.949	13.24	15.40	7.066	5.642	13.23
	M460	57.56	11.30	11.59	29.92	34.84	12.20	13.49	29.89
	M6	28.40	6.323	7.637	39.68	15.27	7.268	15.59	39.61

(f) Tetrahedra, Shunn-Ham

Table B.3: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for triangular and tetrahedral element matrices in double precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Order	Matrix	$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.013	0.011	0.011	0.038	0.023	0.021	0.022	0.038
	M132	0.014	0.013	0.014	0.033	0.018	0.018	0.018	0.038
	M3	0.014	0.013	0.014	0.037	0.018	0.018	0.018	0.037
	M460	0.011	0.012	0.011	0.042	0.022	0.021	0.022	0.042
	M6	0.017	0.016	0.015	0.070	0.026	0.025	0.025	0.070
Order = 2	M0	0.023	0.020	0.019	0.114	0.036	0.033	0.034	0.114
	M132	0.031	0.026	0.027	0.178	0.039	0.037	0.037	0.178
	M3	0.022	0.020	0.021	0.125	0.032	0.030	0.031	0.126
	M460	0.027	0.024	0.024	0.178	0.048	0.043	0.044	0.178
	M6	0.029	0.026	0.026	0.221	0.050	0.045	0.047	0.222
Order = 3	M0	0.039	0.028	0.028	0.238	0.059	0.046	0.047	0.239
	M132	0.093	0.046	0.048	0.451	0.116	0.063	0.072	0.452
	M3	0.033	0.028	0.028	0.238	0.051	0.048	0.047	0.239
	M460	0.073	0.042	0.041	0.470	0.112	0.073	0.078	0.470
	M6	0.047	0.041	0.042	0.470	0.081	0.078	0.077	0.470
Order = 4	M0	0.083	0.041	0.042	0.447	0.118	0.065	0.072	0.447
	M132	0.343	0.069	0.076	1.041	0.378	0.112	0.139	1.040
	M3	0.053	0.039	0.040	0.480	0.079	0.069	0.069	0.480
	M460	0.239	0.065	0.066	1.130	0.261	0.121	0.141	1.130
	M6	0.068	0.061	0.060	0.913	0.133	0.116	0.127	0.913
Order = 5	M0	0.213	0.055	0.057	0.725	0.258	0.088	0.109	0.725
	M132	0.689	0.110	0.140	1.988	0.591	0.392	0.395	1.989
	M3	0.094	0.053	0.053	0.764	0.131	0.092	0.107	0.764
	M460	0.892	0.095	0.095	2.148	0.727	0.195	0.222	2.149
	M6	0.088	0.082	0.083	1.519	0.186	0.156	0.190	1.520
Order = 6	M0	0.298	0.071	0.073	1.100	0.331	0.107	0.149	1.099
	M132	1.225	0.142	0.231	3.626	0.849	0.455	0.654	3.783
	M3	0.173	0.068	0.071	1.124	0.230	0.123	0.158	1.123
	M460	1.552	0.128	0.140	3.484	1.118	0.262	0.506	3.409
	M6	0.142	0.109	0.107	2.131	0.247	0.221	0.263	2.128

(a) Quadrangles, Gauss-Legendre Method

Order	Matrix	$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 2	M0	0.018	0.018	0.019	0.114	0.034	0.032	0.033	0.114
	M132	0.031	0.027	0.027	0.178	0.039	0.036	0.037	0.178
	M3	0.022	0.020	0.020	0.125	0.032	0.030	0.031	0.125
	M460	0.027	0.023	0.024	0.178	0.048	0.043	0.044	0.178
	M6	0.030	0.026	0.026	0.221	0.050	0.045	0.047	0.221
Order = 3	M0	0.025	0.024	0.028	0.239	0.044	0.042	0.047	0.238
	M132	0.088	0.046	0.048	0.451	0.114	0.067	0.072	0.452
	M3	0.034	0.028	0.028	0.238	0.051	0.048	0.047	0.238
	M460	0.064	0.041	0.042	0.469	0.095	0.079	0.077	0.469
	M6	0.043	0.044	0.042	0.470	0.081	0.073	0.077	0.470
Order = 4	M0	0.033	0.032	0.041	0.447	0.056	0.052	0.072	0.447
	M132	0.315	0.070	0.080	1.041	0.190	0.109	0.138	1.042
	M3	0.052	0.039	0.040	0.480	0.078	0.070	0.076	0.480
	M460	0.221	0.066	0.067	1.129	0.258	0.120	0.144	1.130
	M6	0.068	0.061	0.061	0.913	0.133	0.116	0.116	0.913
Order = 5	M0	0.041	0.041	0.057	0.725	0.066	0.062	0.098	0.725
	M132	0.583	0.101	0.151	1.989	0.534	0.205	0.382	1.986
	M3	0.093	0.053	0.053	0.693	0.132	0.094	0.107	0.693
	M460	0.798	0.095	0.112	1.950	0.691	0.191	0.222	1.949
	M6	0.086	0.084	0.083	1.377	0.184	0.162	0.190	1.377
Order = 6	M0	0.049	0.045	0.073	1.100	0.077	0.079	0.160	1.099
	M132	1.118	0.137	0.427	3.785	0.784	0.460	0.641	3.561
	M3	0.114	0.068	0.070	1.124	0.157	0.121	0.147	1.123
	M460	1.399	0.133	0.161	3.466	1.052	0.262	0.421	3.395
	M6	0.141	0.109	0.109	2.129	0.247	0.215	0.274	2.129

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Table B.4: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for quadrilateral element matrices in single precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.032	0.028	0.028	0.202	0.059	0.053	0.058	0.203
	M132	0.033	0.032	0.033	0.176	0.039	0.040	0.048	0.176
	M3	0.033	0.032	0.033	0.176	0.042	0.040	0.043	0.176
	M460	0.028	0.028	0.028	0.202	0.057	0.052	0.053	0.202
	M6	0.045	0.042	0.042	0.512	0.072	0.069	0.080	0.512
Order = 2	M0	0.181	0.076	0.073	1.152	0.247	0.131	0.154	1.306
	M132	0.250	0.100	0.143	1.501	0.324	0.248	0.336	1.694
	M3	0.106	0.073	0.094	1.159	0.154	0.140	0.179	1.133
	M460	0.261	0.093	0.098	1.941	0.352	0.181	0.222	1.905
	M6	0.136	0.122	0.130	3.102	0.332	0.237	0.437	3.383
Order = 3	M0	0.691	0.146	0.159	4.350	0.848	0.387	0.522	4.191
	M132	1.211	0.355	1.479	8.064	1.087	0.610	2.433	7.972
	M3	0.409	0.148	0.988	4.118	0.628	0.475	0.711	4.185
	M460	1.916	0.227	0.319	8.422	1.664	0.719	1.548	8.368
	M6	0.490	0.256	0.883	12.38	1.378	1.218	2.321	12.43
Order = 4	M0	2.074	0.245	1.377	12.53	1.423	0.955	3.666	12.53
	M132	4.855	1.468	7.293	30.48	3.240	1.797	59.15	30.47
	M3	0.948	0.251	2.599	12.29	1.056	0.889	3.671	12.29
	M460	5.236	0.449	2.486	31.10	3.733	2.501	8.829	31.09
	M6	1.168	0.484	1.993	36.50	2.506	2.158	10.28	36.70
Order = 5	M0	5.237	0.398	3.141	30.17	3.476	1.656	8.347	30.16
	M132	13.22	6.706	23.47	89.84	6.948	4.774	315.66	90.08
	M3	2.250	0.490	6.354	30.16	2.177	1.537	8.384	30.20
	M460	15.87	1.342	3.938	89.97	8.864	4.807	24.79	89.90
	M6	2.356	0.936	3.804	90.15	4.546	3.886		89.88
Order = 6	M0		2.913	9.303	65.07	5.999	3.702		65.02
	M132		13.42	60.03	227.08		8.981		226.90
	M3		1.210	14.03	65.15		2.758		65.11
	M460		3.797	15.30	226.67		7.549		226.74
	M6		1.404	6.977	194.24		5.782		194.10

(c) Hexahedra, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 2	M0	0.074	0.071	0.072	1.306	0.140	0.128	0.160	1.307
	M132	0.219	0.110	0.148	1.691	0.308	0.242	0.356	1.695
	M3	0.107	0.074	0.095	1.160	0.155	0.141	0.184	1.134
	M460	0.261	0.094	0.098	1.955	0.353	0.199	0.220	1.903
	M6	0.137	0.121	0.129	3.313	0.332	0.237	0.449	3.378
Order = 3	M0	0.144	0.134	0.162	4.182	0.366	0.359	0.560	4.359
	M132	1.057	0.281	1.901	7.964	1.006	0.584	2.600	7.980
	M3	0.402	0.145	0.913	4.238	0.631	0.486	0.807	4.136
	M460	1.673	0.227	1.094	8.358	1.577	0.713	1.089	8.404
	M6	0.427	0.253	0.817	12.37	1.367	1.190	2.359	12.39
Order = 4	M0	0.298	0.217	1.316	12.53	0.979	0.428	3.629	12.55
	M132	4.625	1.520	8.420	30.39	3.280	1.683	57.89	30.47
	M3	0.801	0.251	2.595	12.29	1.022	0.865	3.523	12.30
	M460	5.062	0.457	3.864	31.12	3.604	2.478	8.830	31.13
	M6	1.169	0.482	1.962	36.53	2.561	2.157	10.33	36.50
Order = 5	M0	0.468	0.324	3.247	30.18	1.321	0.722	8.355	30.15
	M132	11.97	5.998	28.43	90.08	6.607	4.050	328.64	90.05
	M3	2.270	0.436	6.357	30.18	2.175	1.494	8.383	30.22
	M460	12.51	1.259	10.92	89.93	6.981	4.333	24.68	89.98
	M6	1.954	0.938	3.961	90.35	4.148	3.862	24.67	90.28
Order = 6	M0	1.112	0.701	9.141	65.21	2.047	1.999		64.98
	M132		13.35	69.10	227.04	13.27	8.579		226.96
	M3		1.172	14.32	65.24	3.289	2.600		65.15
	M460		3.275	31.66	226.80	13.58	7.236		226.42
	M6		1.350	6.788	194.11	6.603	5.785		194.09

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table B.4: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for hexahedral element matrices in single precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.012	0.008	0.009	0.028	0.018	0.017	0.028	0.028
	M132	0.011	0.010	0.011	0.024	0.014	0.013	0.014	0.024
	M3	0.012	0.011	0.011	0.027	0.015	0.013	0.014	0.027
	M460	0.009	0.009	0.011	0.031	0.016	0.017	0.017	0.031
	M6	0.015	0.012	0.012	0.050	0.021	0.020	0.020	0.050
Order = 2	M0	0.023	0.013	0.014	0.072	0.029	0.025	0.025	0.072
	M132	0.027	0.019	0.019	0.085	0.031	0.025	0.025	0.086
	M3	0.022	0.015	0.015	0.069	0.029	0.022	0.023	0.069
	M460	0.029	0.016	0.017	0.083	0.036	0.030	0.030	0.084
	M6	0.033	0.018	0.019	0.114	0.040	0.033	0.034	0.114
Order = 3	M0	0.052	0.019	0.020	0.122	0.053	0.034	0.034	0.122
	M132	0.146	0.030	0.030	0.205	0.124	0.043	0.043	0.205
	M3	0.052	0.020	0.021	0.131	0.049	0.032	0.032	0.131
	M460	0.082	0.027	0.027	0.201	0.079	0.048	0.048	0.201
	M6	0.066	0.028	0.028	0.232	0.069	0.054	0.050	0.232
Order = 4	M0	0.109	0.027	0.027	0.221	0.089	0.045	0.045	0.221
	M132	0.373	0.044	0.045	0.413	0.255	0.065	0.067	0.413
	M3	0.098	0.027	0.027	0.221	0.085	0.048	0.045	0.221
	M460	0.238	0.040	0.041	0.437	0.169	0.072	0.076	0.437
	M6	0.144	0.039	0.040	0.436	0.130	0.077	0.072	0.436
Order = 5	M0	0.344	0.037	0.040	0.365	0.244	0.063	0.062	0.365
	M132	1.542	0.069	0.070	0.763	0.903	0.127	0.113	0.764
	M3	0.306	0.035	0.034	0.371	0.165	0.065	0.070	0.371
	M460	0.876	0.057	0.058	0.811	0.537	0.116	0.115	0.812
	M6	0.312	0.053	0.054	0.699	0.200	0.104	0.110	0.700
Order = 6	M0	0.588	0.046	0.048	0.560	0.357	0.079	0.078	0.560
	M132	2.761	0.112	0.108	1.233	1.555	0.255	0.189	1.229
	M3	0.450	0.079	0.044	0.531	0.350	0.084	0.085	0.530
	M460	1.739	0.082	0.088	1.368	1.068	0.162	0.164	1.368
	M6	0.668	0.064	0.068	1.052	0.353	0.138	0.150	1.052

(e) Triangles, Williams-Shunn

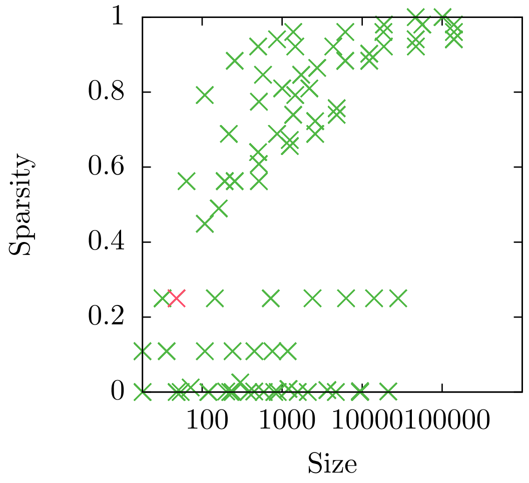
Matrix		$\beta = 0$				$\beta \neq 0$			
		TEX	CSE	SPA	NAI	TEX	CSE	SPA	NAI
Order = 1	M0	0.018	0.015	0.015	0.062	0.032	0.028	0.040	0.062
	M132	0.017	0.017	0.017	0.051	0.021	0.021	0.022	0.051
	M3	0.020	0.017	0.017	0.051	0.023	0.022	0.022	0.051
	M460	0.015	0.015	0.014	0.062	0.028	0.028	0.029	0.062
	M6	0.028	0.021	0.021	0.141	0.039	0.035	0.036	0.141
Order = 2	M0	0.076	0.030	0.030	0.240	0.080	0.061	0.055	0.240
	M132	0.106	0.039	0.041	0.298	0.112	0.054	0.055	0.299
	M3	0.098	0.034	0.035	0.243	0.102	0.047	0.048	0.242
	M460	0.091	0.037	0.035	0.308	0.098	0.065	0.066	0.308
	M6	0.166	0.047	0.049	0.663	0.141	0.091	0.094	0.663
Order = 3	M0	0.665	0.056	0.054	0.715	0.478	0.124	0.109	0.715
	M132	1.393	0.120	0.102	0.944	0.795	0.281	0.155	0.944
	M3	0.802	0.061	0.062	0.655	0.536	0.223	0.093	0.655
	M460	1.026	0.074	0.073	1.070	0.660	0.151	0.159	1.070
	M6	0.928	0.094	0.099	1.934	0.544	0.292	0.201	1.936
Order = 4	M0	3.241	0.110	0.110	1.762	1.206	0.358	0.189	1.763
	M132	4.764	0.296	0.405	2.857	3.084	0.660	0.520	2.740
	M3	2.285	0.151	0.135	1.649	1.477	0.451	0.343	1.561
	M460	6.409	0.185	0.186	2.976	3.667	0.348	0.315	2.856
	M6	4.619	0.216	0.353	4.368	2.901	0.648	1.025	4.686
Order = 5	M0	8.102	0.754	0.520	3.236	4.821	1.144	0.704	3.296
	M132	20.85	1.746	1.805	6.241	15.01	1.949	2.080	6.139
	M3	6.143	0.501	0.534	3.151	3.880	0.891	0.602	3.357
	M460	16.99	0.893	0.962	6.552	8.504	1.345	1.301	6.475
	M6	11.60	1.116	0.841	9.404	5.097	1.822	1.617	9.404
Order = 6	M0	16.45	1.589	0.994	6.595	9.225	1.811	1.249	6.368
	M132	74.86	6.169	3.945	13.68	84.84	6.600	4.463	13.76
	M3	14.34	1.381	1.659	6.381	7.738	1.822	1.941	6.319
	M460	39.84	4.383	3.993	14.11	20.92	4.323	2.894	14.11
	M6	22.74	1.989	2.820	18.88	9.825	3.097	2.988	18.90

(f) Tetrahedra, Shunn-Ham

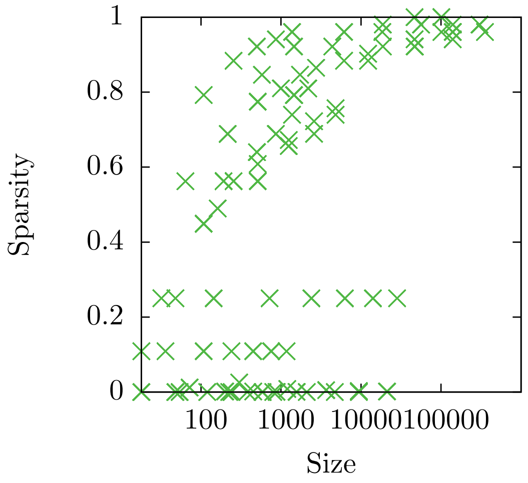
Table B.4: Results of experiments conducted to verify hypothesis stated in Section 4.2. TEX corresponds to kernels using the texture cache (Section 4.2.1), SPA to kernels without sparsity elimnation (Section 4.2.3), CSE to kernels with common sub-expression elimination (Section 4.2.2) and NAI is the naive 3-loop implementation (Section 4.2.4). Results for triangular and tetrahedral element matrices in single precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Appendix C

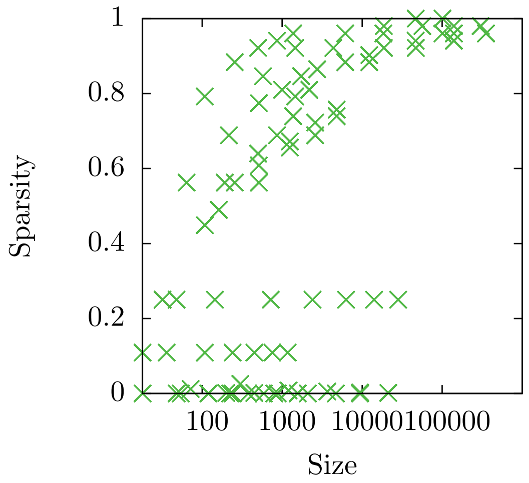
Plots of Results for Individual Optimisations



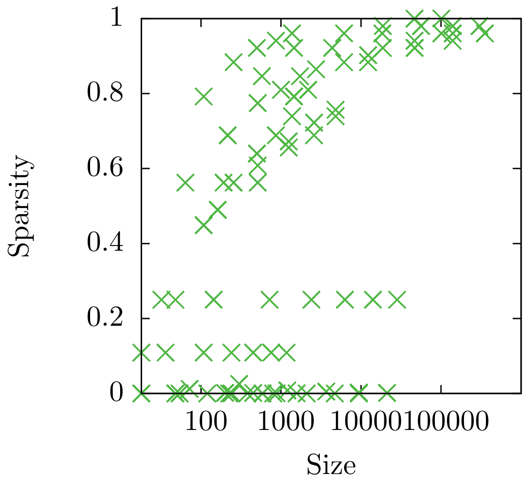
(a) GTX 780 Ti, single precision, $\beta = 0$



(b) Tesla K40c, double precision, $\beta \neq 0$

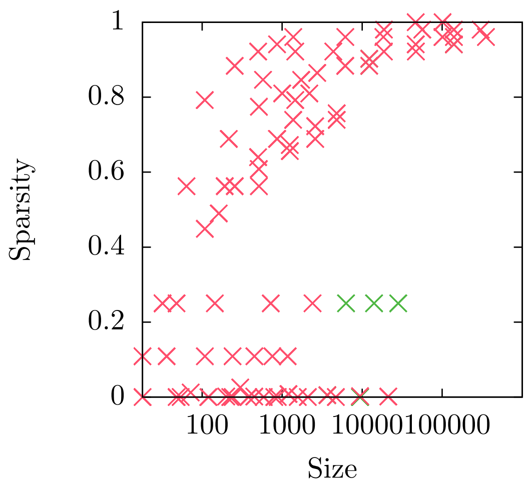


(c) GTX 780 Ti, double precision, $\beta \neq 0$

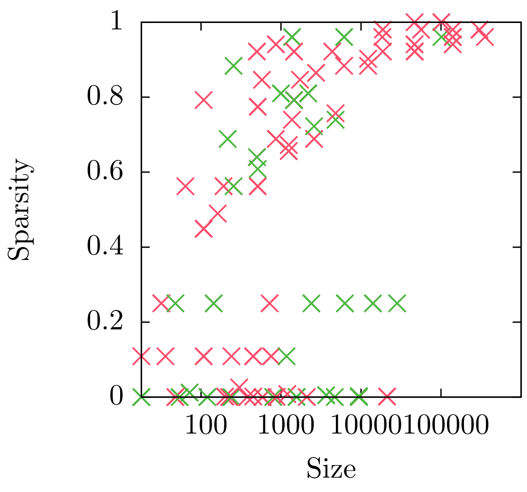


(d) GTX 780 Ti, $\beta \neq 0$

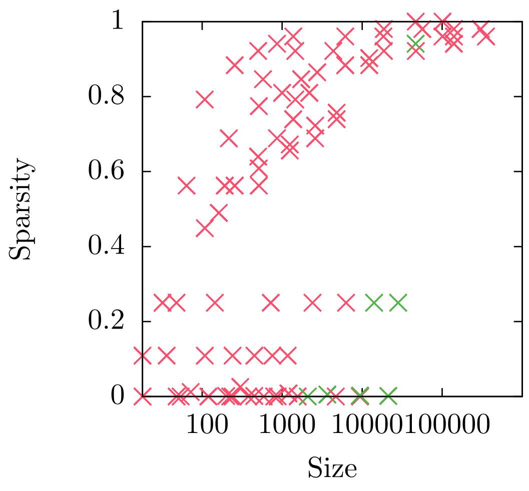
Figure C.1: Comparison of kernels embedding values the operator matrix in the code and those accessing it through the texture unit. Positive and neutral effects of value embedding on the set of benchmark matrices are indicated as green.



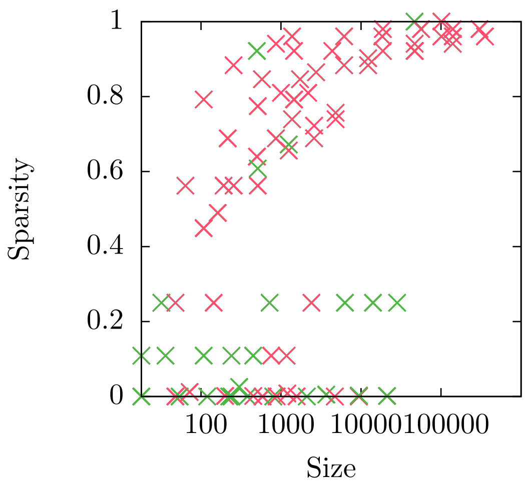
(a) Tesla K40c, single precision, $\beta = 0$



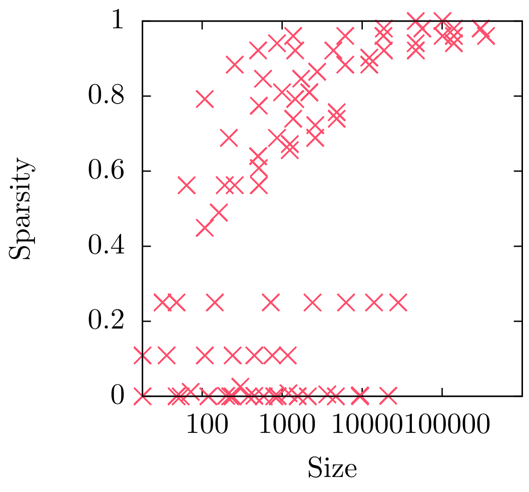
(b) GTX 780 Ti, single precision, $\beta = 0$



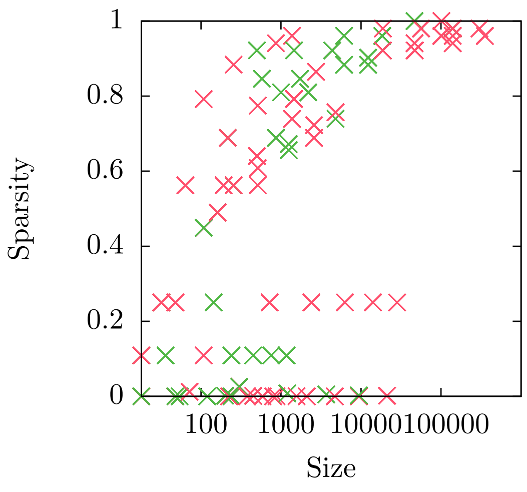
(c) Tesla K40c, double precision, $\beta \neq 0$



(d) GTX 780 Ti, double precision, $\beta \neq 0$

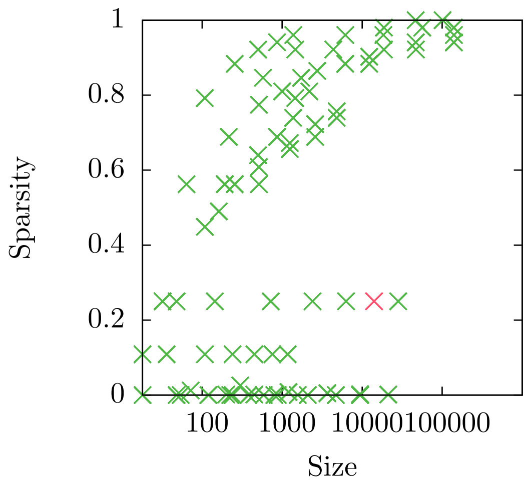


(e) Tesla K40c, single precision, $\beta \neq 0$

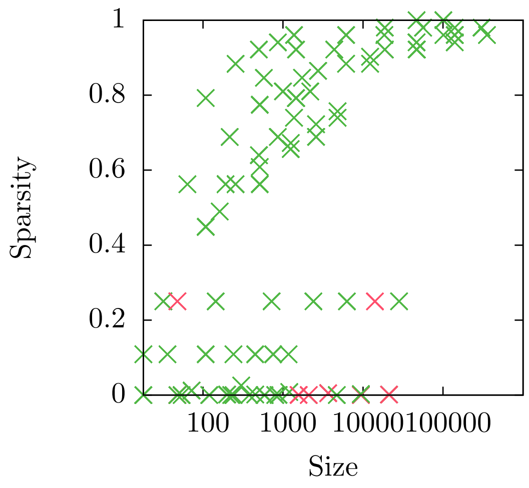


(f) GTX 780 Ti, single precision, $\beta \neq 0$

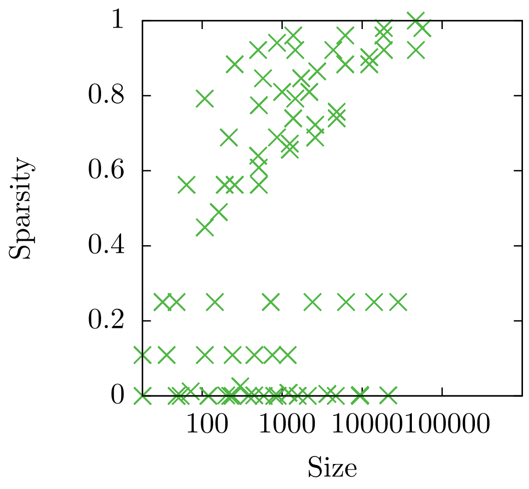
Figure C.2: Comparison of kernels eliminating common sub-expressions from the operator matrix and those performing no such optimisation. Positive effects of common sub-expression elimination on the set of benchmark matrices are indicated as green.



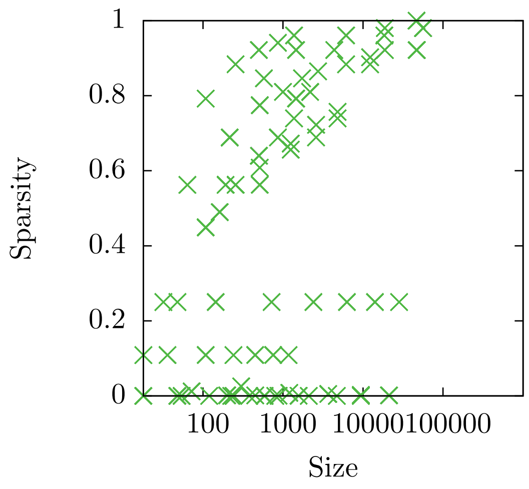
(a) Tesla K40c, single precision, $\beta = 0$



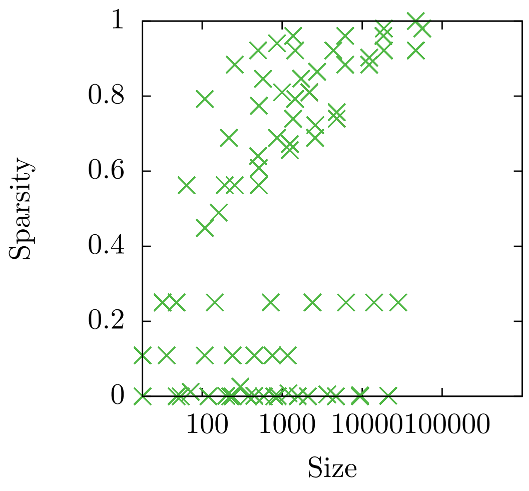
(b) GTX 780 Ti, single precision, $\beta = 0$



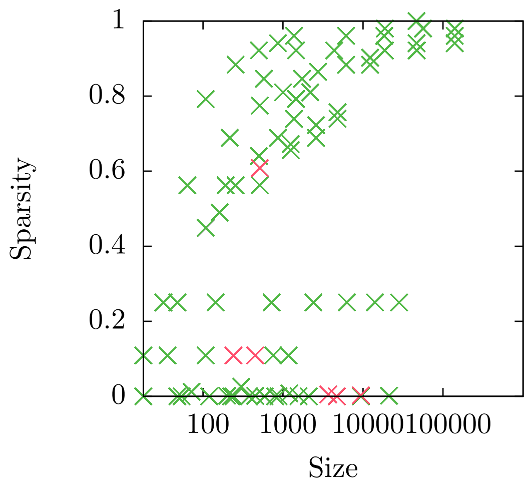
(c) Tesla K40c, double precision, $\beta \neq 0$



(d) GTX 780 Ti, double precision, $\beta \neq 0$

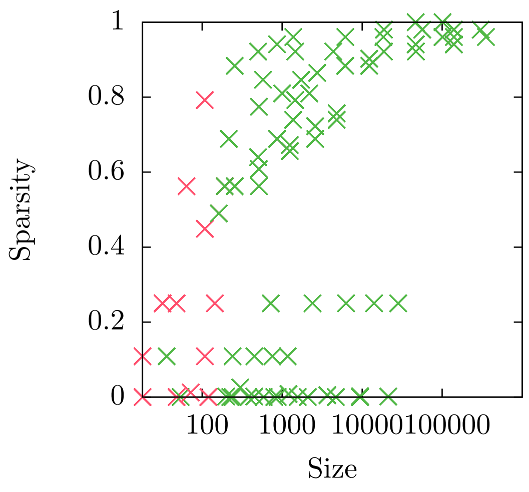


(e) Tesla K40c, single precision, $\beta \neq 0$

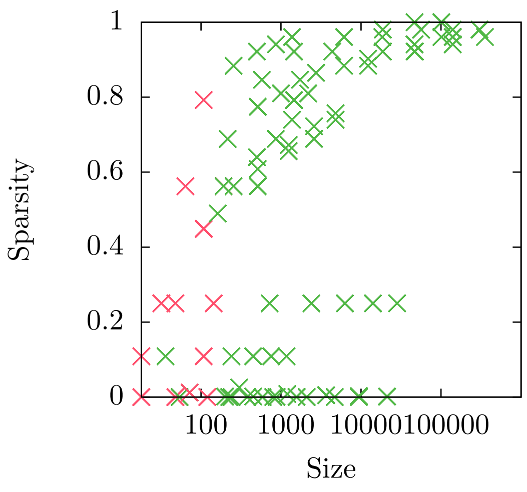


(f) GTX 780 Ti, single precision, $\beta \neq 0$

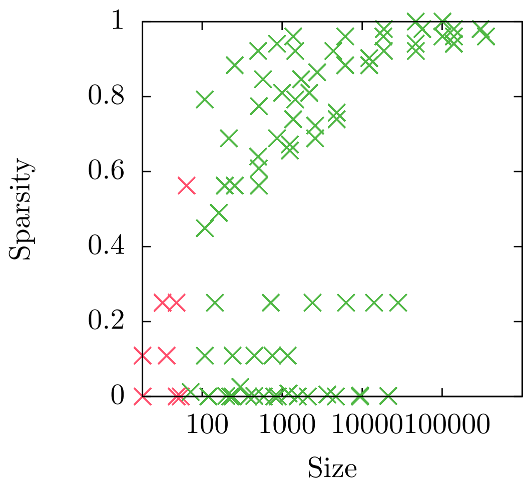
Figure C.3: Comparison of kernels eliminating sparsity from the operator matrix and those performing all the multiplications by zeros. Positive and neutral effects of sparsity elimination on the set of benchmark matrices are indicated as green.



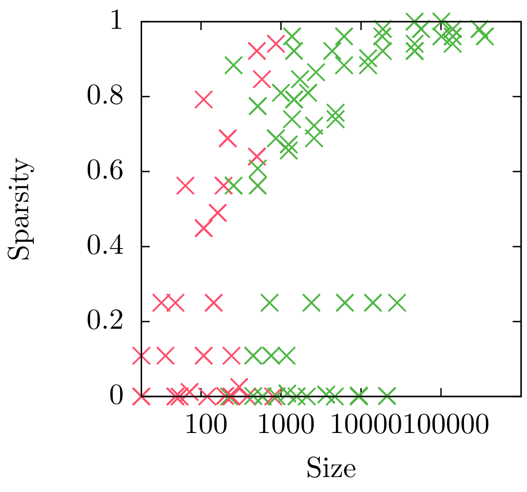
(a) Tesla K40c, single precision, $\beta = 0$



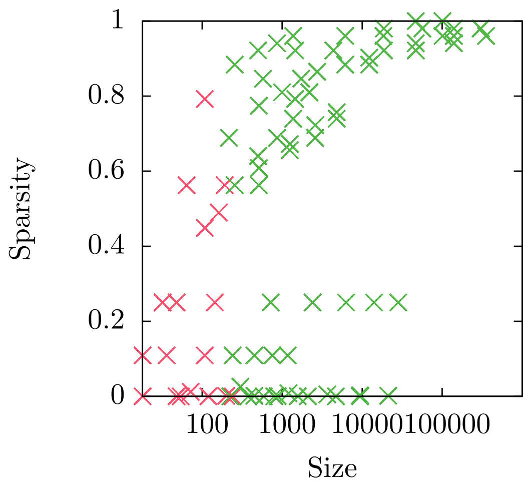
(b) GTX 780 Ti, single precision, $\beta = 0$



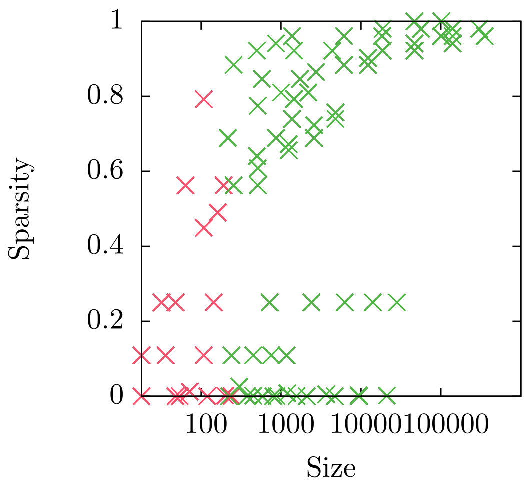
(c) Tesla K40c, double precision, $\beta \neq 0$



(d) GTX 780 Ti, double precision, $\beta \neq 0$



(e) Tesla K40c, single precision, $\beta \neq 0$



(f) GTX 780 Ti, single precision, $\beta \neq 0$

Figure C.4: Comparison of NVIDIA cuBLAS and the naive 3-loop matrix multiplication kernel. Cases when cuBLAS performs better are indicated as green.

Appendix D

Benchmarking Results for GiMMiK Kernels

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.073	0.027	0.064	0.031	0.105	0.047	0.100	0.055
	M132	0.084	0.030	0.093	0.035	0.105	0.040	0.118	0.047
	M3	0.084	0.031	0.093	0.035	0.105	0.040	0.119	0.046
	M460	0.073	0.026	0.066	0.031	0.105	0.047	0.100	0.056
	M6	0.089	0.036	0.102	0.042	0.119	0.057	0.137	0.066
Order = 2	M0	0.123	0.047	0.181	0.053	0.154	0.078	0.218	0.093
	M132	0.161	0.066	0.237	0.071	0.191	0.085	0.292	0.100
	M3	0.123	0.049	0.169	0.056	0.152	0.072	0.201	0.084
	M460	0.215	0.060	0.217	0.068	0.240	0.104	0.262	0.125
	M6	0.218	0.068	0.166	0.074	0.245	0.110	0.209	0.134
Order = 3	M0	0.127	0.073	0.207	0.082	0.171	0.113	0.256	0.132
	M132	0.304	0.115	0.380	0.123	0.321	0.152	0.428	0.184
	M3	0.127	0.073	0.209	0.082	0.171	0.111	0.256	0.133
	M460	0.226	0.109	0.236	0.121	0.266	0.184	0.318	0.221
	M6	0.226	0.108	0.236	0.120	0.267	0.182	0.320	0.221
Order = 4	M0	0.300	0.106	0.455	0.117	0.321	0.155	0.496	0.190
	M132	0.432	0.180	0.651	0.190	0.456	0.259	0.728	0.294
	M3	0.262	0.103	0.280	0.113	0.289	0.161	0.336	0.194
	M460	0.310	0.176	0.936	0.209	0.384	0.304	1.058	0.376
	M6	0.270	0.157	0.531	0.180	0.356	0.273	0.632	0.330
Order = 5	M0	0.346	0.144	0.425	0.155	0.369	0.221	0.471	0.244
	M132	0.527	0.272	1.550	0.285	0.579	0.373	1.684	0.568
	M3	0.269	0.139	0.547	0.151	0.319	0.224	0.658	0.264
	M460	0.680	0.255	1.276	0.290	0.799	0.497	1.483	0.545
	M6	0.522	0.218	0.877	0.247	0.664	0.386	1.077	0.461
Order = 6	M0	0.432	0.189	0.862	0.194	0.459	0.275	0.927	0.310
	M132	0.689	0.361	2.360	0.386	0.753	0.663	2.475	0.790
	M3	0.311	0.181	0.713	0.214	0.383	0.296	0.810	0.377
	M460	0.855	0.356	3.133	0.391	0.997	0.681	3.444	0.766
	M6	0.607	0.288	1.396	0.388	0.760	0.513	1.601	0.677

(a) Quadrangles, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 2	M0	0.123	0.045	0.182	0.054	0.154	0.074	0.219	0.087
	M132	0.161	0.065	0.237	0.071	0.191	0.085	0.293	0.100
	M3	0.123	0.049	0.167	0.056	0.152	0.072	0.202	0.085
	M460	0.215	0.060	0.218	0.068	0.240	0.104	0.263	0.125
	M6	0.218	0.068	0.167	0.074	0.245	0.110	0.209	0.133
Order = 3	M0	0.127	0.063	0.207	0.070	0.171	0.101	0.254	0.122
	M132	0.305	0.116	0.378	0.125	0.321	0.153	0.428	0.184
	M3	0.127	0.072	0.208	0.082	0.171	0.111	0.257	0.132
	M460	0.226	0.110	0.235	0.121	0.266	0.184	0.318	0.220
	M6	0.226	0.110	0.236	0.120	0.266	0.182	0.318	0.220
Order = 4	M0	0.300	0.081	0.453	0.090	0.320	0.128	0.495	0.155
	M132	0.431	0.180	0.655	0.190	0.455	0.259	0.730	0.295
	M3	0.262	0.104	0.282	0.113	0.290	0.160	0.336	0.193
	M460	0.311	0.174	0.936	0.208	0.385	0.304	1.059	0.376
	M6	0.270	0.156	0.529	0.180	0.356	0.273	0.630	0.330
Order = 5	M0	0.346	0.098	0.425	0.110	0.368	0.154	0.473	0.187
	M132	0.527	0.258	1.554	0.284	0.580	0.372	1.680	0.561
	M3	0.269	0.139	0.546	0.161	0.319	0.224	0.659	0.264
	M460	0.680	0.255	1.280	0.292	0.798	0.495	1.486	0.544
	M6	0.522	0.218	0.878	0.246	0.663	0.386	1.078	0.461
Order = 6	M0	0.431	0.120	0.872	0.131	0.459	0.182	0.922	0.220
	M132	0.689	0.359	2.364	0.388	0.755	0.656	2.488	0.780
	M3	0.311	0.185	0.714	0.214	0.383	0.305	0.810	0.377
	M460	0.855	0.355	3.134	0.392	0.996	0.677	3.449	0.764
	M6	0.606	0.287	1.394	0.388	0.760	0.529	1.604	0.676

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Table D.1: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for quadrilateral element matrices in double precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.177	0.071	0.122	0.081	0.212	0.129	0.186	0.155
	M132	0.259	0.079	0.249	0.083	0.269	0.096	0.284	0.110
	M3	0.259	0.078	0.247	0.084	0.269	0.096	0.283	0.110
	M460	0.177	0.070	0.122	0.081	0.212	0.128	0.183	0.155
	M6	0.265	0.108	0.294	0.122	0.293	0.164	0.345	0.198
Order = 2	M0	0.314	0.194	0.939	0.213	0.399	0.365	1.066	0.406
	M132	0.605	0.283	1.397	0.282	0.627	0.399	1.463	0.481
	M3	0.434	0.197	0.700	0.205	0.462	0.274	0.809	0.316
	M460	0.598	0.255	1.419	0.291	0.733	0.516	1.616	0.570
	M6	0.853	0.312	2.043	0.367	0.988	0.504	2.254	0.656
Order = 3	M0	1.186	0.996	4.826	1.134	1.314	1.111	5.066	1.529
	M132	0.657	0.379	2.434	0.423	0.755	0.719	2.630	0.873
	M3	1.524	0.610	4.828	0.768	1.869	1.752	5.377	1.978
	M460	2.018	0.698	7.181	0.882	2.305	1.729	7.775	2.008
	M6	2.530	0.590	10.47	0.715	2.740	1.031	10.99	1.612
Order = 4	M0	2.537	0.700	10.49	0.902	2.729	1.572	10.98	1.878
	M132	4.174	3.175	24.83	4.032	4.462	3.139	25.62	4.352
	M3	1.864	0.652	7.575	0.895	2.130	1.332	7.956	1.728
	M460	4.980	1.442	25.30	1.883	5.405	3.586	26.53	4.087
	M6	5.778	1.276	22.31	1.873	6.189	3.307	23.37	3.940
Order = 5	M0	5.184	4.669	18.39	3.810	5.453	4.985	19.06	5.144
	M132	14.25	7.575	55.09	9.381	14.49	8.035	56.16	9.893
	M3	5.200	1.283	18.39	1.714	5.453	2.410	19.06	3.056
	M460	14.08	5.431	55.13	5.662	14.73	8.935	57.40	9.965
	M6	14.06	1.934	55.11	3.478	14.68	5.674	57.34	6.852
Order = 6	M0	9.923	8.061	54.38	10.08	10.25	10.08	56.23	9.845
	M132	33.35	16.27	185.53	18.54	33.88	15.60	189.87	18.57
	M3	10.27	2.745	39.51	2.731	10.66	4.047	40.72	4.855
	M460	33.68	11.34	183.97	12.16	34.67	17.58	190.04	19.12
	M6	29.17	2.922	118.41	5.531	30.18	8.963	122.30	10.87

(c) Hexahedra, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 2	M0	0.314	0.184	0.939	0.206	0.400	0.311	1.064	0.400
	M132	0.605	0.273	1.409	0.282	0.628	0.393	1.465	0.482
	M3	0.435	0.199	0.702	0.205	0.462	0.274	0.809	0.316
	M460	0.598	0.255	1.417	0.292	0.730	0.517	1.616	0.588
	M6	0.853	0.312	2.041	0.366	0.989	0.504	2.258	0.657
Order = 3	M0	0.948	0.363	2.427	0.419	1.103	0.646	2.702	0.757
	M132	1.185	0.924	4.826	1.103	1.335	1.096	5.065	1.517
	M3	0.656	0.383	2.419	0.423	0.752	0.720	2.621	0.873
	M460	1.531	0.609	4.826	0.780	1.856	1.730	5.373	1.963
	M6	2.020	0.699	7.180	0.881	2.303	1.725	7.784	2.010
Order = 4	M0	0.949	0.385	2.430	0.423	1.103	0.938	2.700	1.072
	M132	4.172	3.032	24.85	3.839	4.439	2.957	25.62	4.162
	M3	1.864	0.653	7.556	0.900	2.138	1.337	7.973	1.731
	M460	4.972	1.382	25.28	1.938	5.416	3.591	26.52	4.100
	M6	5.769	1.274	22.29	1.866	6.186	3.304	23.37	3.953
Order = 5	M0	5.195	0.883	18.38	1.173	5.442	1.939	19.03	2.512
	M132	14.26	6.884	55.10	9.224	14.45	7.258	56.19	9.490
	M3	5.187	1.283	18.38	1.706	5.452	2.418	19.06	3.062
	M460	14.08	4.80	55.16	5.324	14.74	8.393	57.35	9.532
	M6	14.05	1.94	55.12	3.477	14.69	5.679	57.38	6.858
Order = 6	M0	9.920	1.231	54.39	2.006	10.24	2.674	56.26	3.786
	M132	33.36	15.34	185.69	17.56	33.87	14.32	189.87	18.09
	M3	10.29	2.973	39.52	2.813	10.66	4.195	40.67	4.885
	M460	33.68	10.83	183.98	11.47	34.67	16.25	189.94	18.14
	M6	29.18	2.938	118.52	5.751	30.21	8.973	122.24	10.98

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table D.1: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for hexahedral element matrices in double precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.069	0.020	0.077	0.024	0.097	0.036	0.105	0.042
	M132	0.076	0.023	0.065	0.028	0.097	0.031	0.074	0.037
	M3	0.076	0.024	0.067	0.027	0.097	0.031	0.074	0.036
	M460	0.068	0.020	0.079	0.023	0.097	0.036	0.105	0.042
	M6	0.080	0.027	0.114	0.031	0.107	0.044	0.173	0.051
Order = 2	M0	0.081	0.033	0.124	0.040	0.114	0.057	0.174	0.065
	M132	0.122	0.045	0.154	0.049	0.147	0.060	0.195	0.068
	M3	0.121	0.036	0.158	0.041	0.146	0.052	0.220	0.059
	M460	0.082	0.041	0.116	0.046	0.120	0.070	0.154	0.081
	M6	0.123	0.048	0.183	0.053	0.154	0.078	0.218	0.090
Order = 3	M0	0.123	0.050	0.156	0.057	0.155	0.081	0.195	0.091
	M132	0.162	0.075	0.257	0.079	0.193	0.098	0.297	0.113
	M3	0.123	0.050	0.174	0.058	0.153	0.078	0.223	0.090
	M460	0.217	0.067	0.161	0.076	0.244	0.117	0.213	0.140
	M6	0.218	0.073	0.163	0.081	0.247	0.122	0.210	0.145
Order = 4	M0	0.126	0.069	0.284	0.079	0.166	0.108	0.358	0.127
	M132	0.304	0.114	0.387	0.128	0.318	0.157	0.468	0.185
	M3	0.126	0.069	0.285	0.078	0.166	0.108	0.355	0.127
	M460	0.225	0.105	0.301	0.113	0.262	0.176	0.387	0.211
	M6	0.224	0.105	0.303	0.114	0.261	0.174	0.386	0.210
Order = 5	M0	0.262	0.092	0.401	0.104	0.284	0.139	0.446	0.162
	M132	0.388	0.181	0.505	0.199	0.408	0.305	0.553	0.281
	M3	0.256	0.090	0.248	0.102	0.282	0.143	0.302	0.169
	M460	0.268	0.159	0.767	0.175	0.329	0.260	0.902	0.304
	M6	0.265	0.142	0.512	0.153	0.324	0.245	0.664	0.288
Order = 6	M0	0.304	0.121	0.343	0.135	0.328	0.176	0.376	0.222
	M132	0.437	0.290	0.702	0.304	0.466	0.452	0.773	0.443
	M3	0.265	0.116	0.412	0.127	0.296	0.191	0.470	0.219
	M460	0.317	0.271	0.700	0.265	0.407	0.404	0.835	0.501
	M6	0.274	0.188	0.808	0.206	0.379	0.337	0.922	0.375

(e) Triangles, Williams-Shunn

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.074	0.036	0.071	0.041	0.113	0.064	0.112	0.075
	M132	0.118	0.040	0.127	0.045	0.138	0.050	0.155	0.057
	M3	0.118	0.041	0.127	0.046	0.138	0.051	0.155	0.058
	M460	0.074	0.036	0.073	0.042	0.113	0.064	0.113	0.075
	M6	0.124	0.054	0.153	0.064	0.156	0.084	0.195	0.101
Order = 2	M0	0.217	0.078	0.163	0.086	0.248	0.136	0.222	0.163
	M132	0.301	0.100	0.367	0.106	0.314	0.126	0.434	0.147
	M3	0.261	0.085	0.297	0.092	0.273	0.110	0.333	0.124
	M460	0.219	0.092	0.189	0.101	0.256	0.163	0.295	0.196
	M6	0.267	0.126	0.346	0.138	0.302	0.200	0.421	0.256
Order = 3	M0	0.265	0.143	0.481	0.156	0.322	0.247	0.598	0.285
	M132	0.476	0.286	0.686	0.306	0.501	0.495	0.726	0.520
	M3	0.349	0.155	0.465	0.175	0.371	0.267	0.511	0.258
	M460	0.274	0.199	0.538	0.218	0.389	0.344	0.690	0.398
	M6	0.363	0.249	1.020	0.265	0.460	0.493	1.176	0.530
Order = 4	M0	0.360	0.261	1.220	0.282	0.451	0.566	1.346	0.539
	M132	0.730	1.056	2.866	0.984	0.769	1.316	3.075	1.297
	M3	0.482	0.365	1.340	0.389	0.525	0.812	1.402	0.831
	M460	0.690	0.851	2.335	0.800	0.839	0.991	2.622	0.894
	M6	0.947	0.673	2.969	0.545	1.112	0.933	3.147	1.572
Order = 5	M0	0.860	1.260	2.054	1.172	0.999	1.707	2.266	1.689
	M132	1.045	6.020	3.986	5.511	1.177	9.872	4.123	9.168
	M3	0.611	1.315	2.009	1.235	0.693	1.669	2.137	1.606
	M460	1.393	2.480	4.070	2.304	1.715	4.287	4.527	4.090
	M6	1.900	1.883	6.038	1.748	2.122	2.953	6.520	2.942
Order = 6	M0	1.197	2.370	4.080	2.180	1.344	3.097	4.405	3.009
	M132	2.926	22.48	9.055	20.08	3.097	40.34	9.327	37.18
	M3	1.439	2.450	4.065	2.263	1.543	2.987	4.277	2.876
	M460	2.498	6.591	8.375	5.878	2.825	9.276	9.082	8.805
	M6	2.998	3.764	11.11	5.530	3.345	5.590	11.85	5.361

(f) Tetrahedra, Shunn-Ham Method

Table D.1: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for triangular and tetrahedral element matrices in double precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.169	0.014	0.043	0.018	0.237	0.027	0.052	0.034
	M132	0.050	0.017	0.057	0.022	0.090	0.023	0.058	0.029
	M3	0.050	0.017	0.054	0.023	0.090	0.023	0.060	0.030
	M460	0.169	0.014	0.041	0.019	0.235	0.027	0.052	0.034
	M6	0.169	0.019	0.059	0.025	0.240	0.032	0.068	0.042
Order = 2	M0	0.169	0.024	0.105	0.030	0.241	0.042	0.124	0.055
	M132	0.232	0.035	0.135	0.042	0.291	0.047	0.169	0.060
	M3	0.169	0.026	0.082	0.033	0.246	0.039	0.112	0.051
	M460	0.171	0.031	0.132	0.039	0.249	0.055	0.172	0.072
	M6	0.171	0.034	0.083	0.043	0.250	0.059	0.121	0.078
Order = 3	M0	0.172	0.036	0.109	0.047	0.246	0.060	0.127	0.078
	M132	0.233	0.061	0.193	0.070	0.293	0.083	0.206	0.111
	M3	0.171	0.036	0.110	0.048	0.245	0.060	0.127	0.079
	M460	0.173	0.055	0.135	0.071	0.263	0.096	0.166	0.128
	M6	0.173	0.055	0.136	0.072	0.263	0.102	0.166	0.128
Order = 4	M0	0.232	0.055	0.253	0.065	0.291	0.086	0.283	0.116
	M132	0.343	0.095	0.357	0.108	0.387	0.141	0.398	0.174
	M3	0.232	0.052	0.134	0.063	0.291	0.089	0.180	0.113
	M460	0.235	0.087	0.512	0.103	0.341	0.160	0.569	0.227
	M6	0.234	0.082	0.241	0.104	0.338	0.151	0.319	0.190
Order = 5	M0	0.286	0.074	0.207	0.086	0.330	0.113	0.226	0.148
	M132	0.402	0.135	0.760	0.161	0.465	0.255	0.764	0.277
	M3	0.233	0.071	0.283	0.083	0.323	0.121	0.319	0.173
	M460	0.564	0.127	0.632	0.153	0.711	0.245	0.688	0.328
	M6	0.452	0.109	0.442	0.135	0.611	0.210	0.513	0.305
Order = 6	M0	0.343	0.097	0.483	0.110	0.383	0.139	0.515	0.183
	M132	0.518	0.187	1.262	0.208	0.583	0.556	1.337	0.405
	M3	0.235	0.091	0.319	0.106	0.340	0.159	0.388	0.227
	M460	0.677	0.172	1.617	0.265	0.832	0.336	1.787	0.448
	M6	0.455	0.146	0.611	0.180	0.661	0.282	0.771	0.407

(a) Quadrangles, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 2	M0	0.169	0.023	0.104	0.029	0.242	0.041	0.122	0.053
	M132	0.232	0.035	0.135	0.042	0.291	0.047	0.167	0.061
	M3	0.169	0.026	0.081	0.033	0.244	0.039	0.112	0.051
	M460	0.170	0.031	0.132	0.038	0.247	0.056	0.172	0.073
	M6	0.171	0.034	0.081	0.043	0.249	0.059	0.124	0.078
Order = 3	M0	0.171	0.032	0.110	0.040	0.248	0.054	0.128	0.071
	M132	0.233	0.062	0.193	0.069	0.295	0.083	0.205	0.112
	M3	0.171	0.036	0.109	0.047	0.246	0.060	0.128	0.078
	M460	0.173	0.055	0.136	0.071	0.263	0.102	0.168	0.127
	M6	0.173	0.055	0.136	0.071	0.264	0.102	0.165	0.127
Order = 4	M0	0.232	0.041	0.254	0.052	0.290	0.067	0.285	0.090
	M132	0.342	0.094	0.359	0.107	0.387	0.140	0.398	0.174
	M3	0.232	0.052	0.133	0.063	0.291	0.090	0.180	0.113
	M460	0.235	0.087	0.504	0.106	0.341	0.158	0.571	0.226
	M6	0.234	0.082	0.239	0.104	0.338	0.152	0.318	0.190
Order = 5	M0	0.286	0.051	0.206	0.063	0.329	0.081	0.226	0.120
	M132	0.402	0.132	0.761	0.159	0.465	0.253	0.764	0.273
	M3	0.233	0.071	0.286	0.083	0.323	0.121	0.317	0.172
	M460	0.565	0.127	0.630	0.156	0.711	0.245	0.689	0.327
	M6	0.452	0.109	0.443	0.136	0.612	0.210	0.514	0.305
Order = 6	M0	0.342	0.061	0.476	0.074	0.383	0.100	0.508	0.142
	M132	0.518	0.188	1.265	0.209	0.584	0.543	1.337	0.402
	M3	0.235	0.091	0.319	0.106	0.340	0.160	0.390	0.228
	M460	0.677	0.177	1.617	0.263	0.833	0.334	1.790	0.447
	M6	0.455	0.145	0.611	0.179	0.661	0.281	0.771	0.406

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Table D.2: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for quadrilateral element matrices in single precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.171	0.037	0.076	0.045	0.249	0.068	0.098	0.089
	M132	0.232	0.043	0.132	0.049	0.296	0.052	0.136	0.068
	M3	0.232	0.041	0.132	0.048	0.297	0.051	0.137	0.069
	M460	0.171	0.036	0.076	0.044	0.251	0.068	0.098	0.089
	M6	0.232	0.056	0.145	0.068	0.292	0.091	0.169	0.128
Order = 2	M0	0.236	0.098	0.508	0.114	0.347	0.170	0.574	0.244
	M132	0.459	0.132	0.760	0.156	0.500	0.310	0.833	0.240
	M3	0.343	0.100	0.385	0.117	0.387	0.174	0.458	0.186
	M460	0.453	0.126	0.754	0.154	0.610	0.239	0.857	0.343
	M6	0.676	0.162	1.090	0.222	0.812	0.302	1.203	0.386
Order = 3	M0	0.678	0.194	1.166	0.275	0.809	0.484	1.239	0.520
	M132	0.812	0.445	2.339	0.471	0.880	0.930	2.333	0.973
	M3	0.464	0.191	1.172	0.230	0.543	0.594	1.204	0.623
	M460	1.048	0.305	2.304	0.466	1.456	0.890	2.441	1.024
	M6	1.240	0.348	3.403	0.537	1.646	1.489	3.495	1.529
Order = 4	M0	1.322	0.337	5.572	0.458	1.798	1.296	5.699	1.523
	M132	2.997	1.709	13.12	1.390	3.138	1.868	13.27	2.246
	M3	1.372	0.341	3.992	0.467	1.491	1.122	4.179	1.370
	M460	2.898	0.616	13.20	0.866	3.747	3.041	13.64	3.448
	M6	3.290	0.647	11.70	0.896	4.092	2.879	12.20	3.333
Order = 5	M0	2.719	0.542	8.546	0.750	3.256	2.209	8.465	2.286
	M132	6.767	9.165	25.54	3.295	7.292	3.756	24.82	4.653
	M3	2.713	0.614	8.560	0.750	3.240	1.967	8.461	2.292
	M460	7.194	1.676	25.77	1.646	8.577	5.883	25.48	6.086
	M6	7.173	1.220	25.76	1.791	8.571	5.175	25.46	5.704
Order = 6	M0	5.030	3.578	28.16	2.630	5.796	4.795	28.69	4.323
	M132	16.12	18.50	96.34	6.905	16.91	9.459	96.77	9.469
	M3	5.427	1.551	20.70	1.291	6.223	3.369	21.27	3.622
	M460	15.89	4.738	95.23	5.039	18.18	10.21	96.81	11.83
	M6	14.07	1.821	61.72	2.172	16.37	7.719	63.43	9.012

(c) Hexahedra, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 2	M0	0.236	0.095	0.509	0.115	0.347	0.167	0.573	0.243
	M132	0.459	0.138	0.762	0.158	0.501	0.309	0.827	0.240
	M3	0.342	0.101	0.387	0.117	0.388	0.175	0.459	0.199
	M460	0.453	0.126	0.757	0.182	0.609	0.256	0.856	0.395
	M6	0.677	0.161	1.089	0.223	0.812	0.302	1.202	0.386
Order = 3	M0	0.678	0.183	1.167	0.284	0.809	0.447	1.238	0.501
	M132	0.811	0.401	2.342	0.467	0.880	0.758	2.335	0.959
	M3	0.464	0.191	1.175	0.230	0.543	0.597	1.206	0.667
	M460	1.053	0.304	2.304	0.473	1.460	0.882	2.441	1.015
	M6	1.246	0.354	3.407	0.537	1.647	1.482	3.495	1.436
Order = 4	M0	1.336	0.297	5.530	0.431	1.797	0.545	5.715	0.851
	M132	2.996	1.724	13.10	1.349	3.123	1.810	13.27	2.227
	M3	1.374	0.336	3.988	0.466	1.493	1.126	4.170	1.371
	M460	2.896	0.620	13.19	0.875	3.751	3.018	13.63	3.440
	M6	3.304	0.648	11.69	0.894	4.087	2.878	12.21	3.339
Order = 5	M0	2.704	0.439	8.553	0.649	3.241	0.991	8.481	2.039
	M132	6.755	7.950	25.55	3.149	7.296	3.669	24.83	4.574
	M3	2.708	0.624	8.556	0.747	3.235	1.969	8.464	2.282
	M460	7.203	1.581	25.82	1.554	8.576	5.731	25.47	6.032
	M6	7.171	1.219	25.76	1.799	8.563	5.244	25.46	5.695
Order = 6	M0	5.039	0.907	28.17	0.915	5.769	2.447	28.66	2.775
	M132	16.11	18.24	96.31	6.703	16.91	8.815	97.00	9.718
	M3	5.427	1.526	20.73	1.294	6.216	3.394	21.26	3.632
	M460	15.87	4.168	95.36	4.558	18.19	9.726	96.78	11.06
	M6	14.09	1.775	61.75	2.457	16.35	7.725	63.42	9.097

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table D.2: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for hexahedral element matrices in single precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.046	0.011	0.047	0.015	0.091	0.021	0.085	0.025
	M132	0.046	0.013	0.040	0.017	0.075	0.017	0.045	0.021
	M3	0.047	0.014	0.041	0.017	0.075	0.017	0.045	0.022
	M460	0.046	0.011	0.047	0.016	0.091	0.021	0.085	0.026
	M6	0.051	0.015	0.074	0.020	0.096	0.025	0.116	0.032
Order = 2	M0	0.169	0.017	0.071	0.022	0.241	0.032	0.102	0.041
	M132	0.071	0.024	0.077	0.029	0.114	0.033	0.118	0.042
	M3	0.070	0.019	0.110	0.025	0.113	0.029	0.157	0.036
	M460	0.169	0.020	0.066	0.026	0.241	0.039	0.081	0.050
	M6	0.169	0.024	0.105	0.030	0.245	0.043	0.125	0.054
Order = 3	M0	0.169	0.025	0.086	0.032	0.244	0.044	0.105	0.056
	M132	0.231	0.039	0.121	0.046	0.294	0.056	0.167	0.067
	M3	0.170	0.028	0.087	0.034	0.248	0.042	0.134	0.053
	M460	0.171	0.035	0.096	0.042	0.251	0.062	0.125	0.081
	M6	0.171	0.037	0.087	0.047	0.251	0.070	0.109	0.084
Order = 4	M0	0.171	0.035	0.167	0.046	0.245	0.058	0.228	0.074
	M132	0.232	0.060	0.229	0.070	0.293	0.085	0.295	0.111
	M3	0.171	0.035	0.167	0.045	0.244	0.058	0.232	0.074
	M460	0.173	0.052	0.171	0.064	0.260	0.094	0.233	0.120
	M6	0.173	0.052	0.169	0.062	0.259	0.100	0.233	0.120
Order = 5	M0	0.232	0.048	0.225	0.059	0.291	0.081	0.250	0.094
	M132	0.287	0.087	0.290	0.110	0.329	0.126	0.317	0.162
	M3	0.232	0.045	0.142	0.057	0.287	0.077	0.174	0.097
	M460	0.234	0.075	0.421	0.089	0.330	0.149	0.501	0.170
	M6	0.233	0.071	0.288	0.083	0.328	0.136	0.373	0.162
Order = 6	M0	0.233	0.064	0.157	0.077	0.292	0.101	0.187	0.135
	M132	0.343	0.156	0.364	0.174	0.385	0.201	0.378	0.292
	M3	0.232	0.058	0.232	0.071	0.290	0.109	0.263	0.124
	M460	0.236	0.113	0.347	0.130	0.350	0.202	0.399	0.281
	M6	0.236	0.091	0.436	0.105	0.348	0.178	0.503	0.252

(e) Triangles, Williams-Shunn

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.169	0.018	0.046	0.024	0.236	0.037	0.058	0.045
	M132	0.067	0.022	0.073	0.027	0.105	0.028	0.076	0.035
	M3	0.067	0.022	0.074	0.028	0.105	0.028	0.074	0.036
	M460	0.169	0.018	0.044	0.023	0.235	0.036	0.059	0.046
	M6	0.170	0.028	0.087	0.036	0.241	0.046	0.096	0.059
Order = 2	M0	0.171	0.039	0.095	0.048	0.250	0.072	0.126	0.092
	M132	0.232	0.053	0.210	0.062	0.298	0.069	0.262	0.090
	M3	0.232	0.044	0.139	0.052	0.297	0.062	0.185	0.081
	M460	0.172	0.045	0.119	0.055	0.258	0.086	0.184	0.111
	M6	0.233	0.063	0.162	0.076	0.294	0.110	0.230	0.154
Order = 3	M0	0.233	0.071	0.252	0.084	0.326	0.141	0.288	0.193
	M132	0.345	0.120	0.324	0.149	0.389	0.255	0.334	0.236
	M3	0.286	0.082	0.224	0.100	0.329	0.118	0.240	0.156
	M460	0.235	0.097	0.280	0.112	0.353	0.196	0.337	0.272
	M6	0.291	0.126	0.508	0.144	0.394	0.241	0.554	0.305
Order = 4	M0	0.290	0.143	0.654	0.161	0.392	0.235	0.720	0.322
	M132	0.517	0.476	1.452	0.390	0.574	0.604	1.591	0.580
	M3	0.345	0.176	0.577	0.197	0.414	0.407	0.672	0.360
	M460	0.566	0.242	1.236	0.294	0.750	0.440	1.376	0.515
	M6	0.681	0.308	1.289	0.277	0.844	0.761	1.442	0.692
Order = 5	M0	0.678	0.641	0.969	0.500	0.809	0.839	1.032	0.714
	M132	0.757	2.239	1.868	2.112	0.824	2.486	1.853	2.456
	M3	0.464	0.671	0.936	0.603	0.534	0.795	0.960	0.668
	M460	0.851	1.194	1.941	0.926	1.421	1.650	2.056	1.345
	M6	1.064	1.741	2.862	0.858	1.590	1.392	2.935	1.372
Order = 6	M0	0.911	1.241	1.941	1.089	1.048	1.479	1.986	1.359
	M132	2.083	5.132	4.169	4.778	2.185	5.569	4.107	5.231
	M3	1.022	2.248	1.883	2.140	1.142	1.466	1.908	1.345
	M460	1.608	5.498	3.972	5.320	2.177	3.368	4.095	3.077
	M6	1.789	3.354	5.251	3.099	2.354	2.457	5.319	4.430

(f) Tetrahedra, Shunn-Ham Method

Table D.2: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for triangular and tetrahedral element matrices in single precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.120	0.020	0.105	0.024	0.132	0.036	0.116	0.038
	M132	0.182	0.024	0.165	0.033	0.192	0.032	0.190	0.037
	M3	0.182	0.024	0.174	0.030	0.192	0.032	0.183	0.040
	M460	0.120	0.020	0.099	0.024	0.132	0.036	0.123	0.046
	M6	0.183	0.027	0.168	0.041	0.195	0.044	0.186	0.053
Order = 2	M0	0.317	0.036	0.247	0.044	0.329	0.061	0.272	0.078
	M132	0.450	0.051	0.391	0.059	0.460	0.066	0.428	0.081
	M3	0.317	0.040	0.242	0.047	0.329	0.057	0.268	0.067
	M460	0.646	0.046	0.260	0.058	0.658	0.081	0.289	0.098
	M6	0.646	0.051	0.232	0.064	0.656	0.084	0.265	0.108
Order = 3	M0	0.318	0.058	0.302	0.069	0.330	0.087	0.328	0.107
	M132	1.155	0.095	0.520	0.104	1.164	0.124	0.595	0.165
	M3	0.318	0.060	0.274	0.070	0.330	0.085	0.326	0.107
	M460	0.649	0.087	0.281	0.096	0.665	0.142	0.347	0.180
	M6	0.649	0.082	0.280	0.084	0.664	0.137	0.346	0.171
Order = 4	M0	1.154	0.087	0.547	0.099	1.171	0.120	0.513	0.139
	M132	1.896	0.145	0.839	0.168	1.916	0.225	0.866	0.245
	M3	0.807	0.075	0.335	0.093	0.911	0.121	0.366	0.139
	M460	1.049	0.134	0.950	0.156	1.201	0.242	1.167	0.318
	M6	0.809	0.108	0.641	0.120	0.929	0.206	0.778	0.258
Order = 5	M0	1.399	0.115	0.565	0.124	1.234	0.170	0.604	0.188
	M132	2.410	0.282	2.446	0.321	2.126	0.315	2.219	0.492
	M3	0.810	0.098	0.855	0.132	0.808	0.153	0.804	0.191
	M460	2.484	0.224	1.663	0.250	2.442	0.400	1.741	0.431
	M6	1.547	0.148	1.131	0.165	1.798	0.285	1.206	0.318
Order = 6	M0	1.872	0.143	0.860	0.163	1.674	0.210	0.908	0.238
	M132	3.202	0.402	3.093	0.450	3.009	0.614	3.116	0.701
	M3	1.023	0.127	0.874	0.146	1.055	0.201	1.036	0.308
	M460	3.276	0.367	3.347	0.397	3.687	0.574	3.527	0.632
	M6	2.276	0.198	1.734	0.230	2.123	0.336	1.808	0.477

(a) Quadrangles, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 2	M0	0.318	0.034	0.256	0.040	0.329	0.056	0.281	0.066
	M132	0.450	0.051	0.401	0.062	0.460	0.066	0.382	0.076
	M3	0.317	0.040	0.242	0.052	0.329	0.056	0.243	0.062
	M460	0.647	0.046	0.267	0.057	0.658	0.081	0.263	0.093
	M6	0.647	0.051	0.234	0.058	0.657	0.084	0.233	0.093
Order = 3	M0	0.318	0.048	0.304	0.060	0.330	0.076	0.328	0.093
	M132	1.155	0.092	0.518	0.100	1.163	0.122	0.597	0.157
	M3	0.318	0.060	0.274	0.067	0.330	0.086	0.326	0.106
	M460	0.650	0.085	0.289	0.089	0.666	0.141	0.346	0.179
	M6	0.649	0.083	0.281	0.082	0.665	0.137	0.353	0.172
Order = 4	M0	1.156	0.061	0.546	0.068	1.170	0.097	0.527	0.110
	M132	1.897	0.152	0.843	0.164	1.915	0.220	0.861	0.238
	M3	0.897	0.083	0.335	0.086	0.910	0.121	0.357	0.141
	M460	1.164	0.139	0.950	0.151	1.201	0.239	1.032	0.281
	M6	0.898	0.118	0.640	0.123	0.930	0.203	0.757	0.248
Order = 5	M0	1.400	0.075	0.582	0.075	1.402	0.115	0.600	0.128
	M132	2.409	0.223	2.171	0.297	2.384	0.297	2.219	0.474
	M3	0.813	0.098	0.757	0.113	0.827	0.155	0.802	0.191
	M460	2.484	0.203	1.663	0.228	2.510	0.389	1.746	0.417
	M6	1.588	0.149	1.131	0.162	1.582	0.254	1.367	0.357
Order = 6	M0	1.893	0.091	0.864	0.087	1.913	0.137	0.904	0.148
	M132	3.142	0.387	3.094	0.418	3.424	0.604	3.128	0.679
	M3	1.048	0.123	0.873	0.135	1.077	0.211	1.035	0.309
	M460	3.340	0.339	3.589	0.374	3.439	0.551	3.534	0.612
	M6	2.008	0.192	1.735	0.225	2.074	0.348	1.810	0.478

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Table D.3: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for quadrilateral element matrices in double precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.400	0.054	0.177	0.064	0.410	0.098	0.194	0.120
	M132	0.898	0.059	0.431	0.066	0.901	0.073	0.408	0.080
	M3	0.898	0.060	0.397	0.066	0.902	0.074	0.406	0.079
	M460	0.400	0.053	0.165	0.056	0.410	0.098	0.176	0.109
	M6	0.900	0.082	0.407	0.085	0.915	0.124	0.415	0.136
Order = 2	M0	1.164	0.143	0.956	0.136	1.062	0.267	1.034	0.292
	M132	2.913	0.239	1.388	0.226	2.562	0.308	1.443	0.378
	M3	1.901	0.145	0.906	0.149	1.687	0.196	0.946	0.233
	M460	2.274	0.195	1.421	0.185	2.249	0.384	1.540	0.408
	M6	3.443	0.210	2.969	0.237	3.484	0.332	2.681	0.460
Order = 3	M0	4.117	0.266	3.295	0.340	3.812	0.735	2.990	0.828
	M132	5.445	0.744	5.745	0.840	5.494	0.887	5.811	1.164
	M3	3.184	0.275	2.896	0.345	2.835	0.578	2.953	0.686
	M460	5.666	0.418	5.792	0.483	5.777	1.401	5.955	1.547
	M6	8.280	0.514	8.624	0.487	8.386	1.302	8.771	1.494
Order = 4	M0	10.78	0.590	10.23	0.658	10.84	1.239	10.54	1.455
	M132	20.64	2.330	24.10	3.126	20.79	2.398	24.40	3.182
	M3	8.608	0.489	9.288	0.688	8.562	1.083	9.382	1.341
	M460	21.37	1.101	24.50	1.238	21.52	2.908	25.34	3.219
	M6	25.34	0.834	27.70	1.230	25.53	2.516	27.98	2.919
Order = 5	M0	23.99	3.604	22.47	2.657	24.13	3.703	22.66	3.730
	M132	70.93	5.643	67.15	6.503	71.01	6.130	67.29	7.138
	M3	23.99	1.123	22.48	1.222	24.11	1.935	22.66	2.327
	M460	65.79	4.448	67.33	4.399	66.05	7.010	67.93	7.647
	M6	65.75	1.438	67.29	2.273	66.01	4.295	67.89	5.062
Order = 6	M0	47.34	7.421	54.07	5.720	47.46	7.472	55.35	7.087
	M132	168.10	12.05	178.86	12.95	168.36	11.85	182.21	13.43
	M3	48.77	2.249	49.20	2.190	48.93	3.231	49.34	3.689
	M460	160.74	8.777	179.03	9.137	161.18	13.74	183.17	14.55
	M6	137.94	2.179	147.41	3.320	138.48	6.780	147.95	8.051

(c) Hexahedra, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 2	M0	1.163	0.140	0.953	0.137	1.207	0.230	1.091	0.291
	M132	2.916	0.202	1.394	0.226	2.938	0.344	1.445	0.379
	M3	1.712	0.129	0.904	0.149	1.925	0.219	0.944	0.234
	M460	2.048	0.174	1.424	0.182	2.349	0.378	1.543	0.422
	M6	3.269	0.209	2.618	0.237	3.377	0.332	2.684	0.460
Order = 3	M0	3.867	0.240	2.913	0.276	3.815	0.440	2.991	0.520
	M132	5.438	0.668	5.917	0.849	5.498	0.853	5.805	1.130
	M3	3.038	0.277	2.896	0.344	2.838	0.580	2.957	0.686
	M460	5.753	0.393	5.792	0.452	5.779	1.364	5.951	1.519
	M6	8.283	0.513	8.626	0.486	8.459	1.304	8.775	1.493
Order = 4	M0	10.75	0.392	10.22	0.417	10.85	0.692	10.55	1.168
	M132	20.78	2.474	23.85	2.615	20.80	2.219	24.41	3.016
	M3	8.491	0.539	9.292	0.691	8.566	1.087	9.374	1.339
	M460	21.35	1.042	24.50	1.361	21.54	2.876	25.32	3.190
	M6	25.33	0.939	27.71	1.096	25.55	2.501	27.99	2.927
Order = 5	M0	23.95	0.644	22.48	0.737	24.07	1.401	22.67	1.804
	M132	70.98	5.155	67.54	6.418	71.02	5.456	67.34	6.824
	M3	24.00	1.104	22.46	1.212	24.11	1.942	22.66	2.334
	M460	65.80	3.979	67.29	4.187	66.05	6.566	67.90	7.251
	M6	65.76	1.440	67.28	2.268	66.06	4.300	67.85	5.055
Order = 6	M0	47.29	0.895	54.06	1.447	47.37	1.928	55.34	2.693
	M132	168.12	11.34	178.77	12.21	168.35	10.77	182.18	13.07
	M3	48.76	2.294	49.19	2.231	48.93	3.319	49.36	3.696
	M460	160.70	8.336	178.97	8.448	161.14	12.66	183.15	13.72
	M6	137.97	2.192	147.34	3.347	138.45	6.799	147.95	8.088

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table D.3: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for hexahedral element matrices in double precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.104	0.016	0.094	0.019	0.118	0.028	0.124	0.030
	M132	0.150	0.019	0.069	0.028	0.159	0.025	0.075	0.026
	M3	0.150	0.020	0.079	0.025	0.160	0.026	0.083	0.037
	M460	0.104	0.015	0.110	0.019	0.118	0.028	0.145	0.033
	M6	0.151	0.023	0.201	0.036	0.163	0.035	0.240	0.041
Order = 2	M0	0.151	0.035	0.188	0.049	0.163	0.050	0.233	0.059
	M132	0.317	0.046	0.255	0.059	0.328	0.057	0.282	0.068
	M3	0.317	0.039	0.259	0.053	0.328	0.049	0.303	0.056
	M460	0.151	0.043	0.178	0.056	0.164	0.059	0.204	0.071
	M6	0.317	0.046	0.249	0.063	0.328	0.065	0.272	0.078
Order = 3	M0	0.317	0.070	0.245	0.085	0.328	0.088	0.268	0.103
	M132	0.450	0.111	0.383	0.143	0.461	0.125	0.418	0.156
	M3	0.317	0.071	0.254	0.094	0.329	0.088	0.286	0.105
	M460	0.647	0.107	0.249	0.130	0.656	0.134	0.280	0.161
	M6	0.646	0.089	0.235	0.112	0.656	0.115	0.268	0.142
Order = 4	M0	0.318	0.128	0.346	0.157	0.330	0.144	0.415	0.174
	M132	1.153	0.245	0.561	0.247	1.159	0.269	0.587	0.270
	M3	0.318	0.128	0.316	0.140	0.330	0.145	0.362	0.155
	M460	0.649	0.233	0.316	0.251	0.663	0.271	0.351	0.278
	M6	0.649	0.165	0.318	0.182	0.664	0.197	0.353	0.212
Order = 5	M0	0.898	0.203	0.478	0.250	0.909	0.228	0.514	0.272
	M132	1.646	0.517	0.730	0.479	1.649	0.566	0.754	0.522
	M3	0.897	0.205	0.352	0.222	0.909	0.233	0.382	0.248
	M460	0.901	0.504	0.848	0.472	0.923	0.541	0.933	0.549
	M6	0.901	0.272	0.772	0.317	0.919	0.322	0.746	0.337
Order = 6	M0	1.157	0.351	0.463	0.322	1.167	0.355	0.484	0.362
	M132	1.901	0.823	0.870	0.774	1.917	0.889	0.900	0.836
	M3	0.809	0.287	0.424	0.331	0.835	0.321	0.457	0.360
	M460	1.051	0.843	0.996	0.802	1.109	0.866	0.934	0.926
	M6	0.812	0.404	0.937	0.455	0.857	0.465	0.950	0.522

(e) Triangles, Williams-Shunn

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.108	0.026	0.097	0.036	0.118	0.045	0.112	0.053
	M132	0.286	0.029	0.204	0.039	0.287	0.036	0.230	0.045
	M3	0.286	0.034	0.242	0.047	0.287	0.041	0.250	0.059
	M460	0.108	0.025	0.105	0.032	0.118	0.044	0.129	0.059
	M6	0.317	0.048	0.235	0.065	0.290	0.061	0.264	0.085
Order = 2	M0	0.645	0.121	0.250	0.148	0.579	0.138	0.281	0.186
	M132	1.153	0.150	0.548	0.159	1.160	0.163	0.584	0.181
	M3	0.894	0.141	0.407	0.148	0.900	0.151	0.435	0.168
	M460	0.646	0.127	0.240	0.139	0.663	0.172	0.278	0.184
	M6	0.900	0.191	0.410	0.202	0.914	0.230	0.442	0.273
Order = 3	M0	0.898	0.421	0.658	0.413	0.918	0.469	0.783	0.533
	M132	2.157	0.598	0.929	0.572	2.185	0.755	0.971	0.688
	M3	1.264	0.408	0.620	0.422	1.406	0.496	0.642	0.459
	M460	0.811	0.567	0.666	0.552	0.933	0.631	0.707	0.647
	M6	1.270	0.608	1.244	0.575	1.428	0.771	1.332	0.739
Order = 4	M0	1.406	1.225	1.200	0.972	1.427	1.299	1.291	1.130
	M132	3.596	1.834	3.401	1.742	3.684	2.018	3.556	1.934
	M3	1.941	1.073	1.807	1.015	1.974	1.243	1.825	1.190
	M460	2.425	1.784	2.375	1.832	2.521	1.923	2.554	1.924
	M6	4.101	1.605	3.629	1.481	3.810	1.688	3.669	1.951
Order = 5	M0	3.364	2.313	2.890	2.484	3.350	2.636	2.635	2.553
	M132	4.764	6.060	5.049	5.538	5.132	8.389	5.108	7.728
	M3	2.915	2.456	2.542	2.266	2.592	2.636	2.597	2.548
	M460	5.031	4.779	5.086	4.593	5.111	5.689	5.254	5.664
	M6	7.551	3.521	7.573	3.386	7.528	4.386	7.730	4.235
Order = 6	M0	5.004	4.661	5.083	4.334	5.202	5.302	5.196	5.050
	M132	14.16	18.47	11.25	16.19	14.19	30.61	11.39	28.18
	M3	6.313	4.682	5.075	4.376	6.320	5.100	5.159	4.893
	M460	9.998	11.42	10.15	11.09	10.10	12.76	10.37	12.65
	M6	12.70	7.043	13.44	7.419	12.79	8.355	13.70	8.023

(f) Tetrahedra, Shunn-Ham

Table D.3: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for triangular and tetrahedral element matrices in double precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.143	0.011	0.033	0.014	0.199	0.021	0.048	0.029
	M132	0.042	0.013	0.047	0.016	0.075	0.018	0.048	0.027
	M3	0.042	0.013	0.049	0.017	0.075	0.018	0.054	0.023
	M460	0.143	0.011	0.038	0.015	0.200	0.021	0.044	0.029
	M6	0.143	0.015	0.053	0.020	0.203	0.025	0.059	0.032
Order = 2	M0	0.144	0.018	0.090	0.025	0.204	0.033	0.110	0.049
	M132	0.195	0.027	0.121	0.036	0.244	0.036	0.150	0.050
	M3	0.144	0.019	0.071	0.026	0.206	0.030	0.095	0.039
	M460	0.145	0.023	0.123	0.029	0.210	0.042	0.153	0.056
	M6	0.145	0.026	0.071	0.033	0.209	0.045	0.104	0.066
Order = 3	M0	0.146	0.028	0.095	0.037	0.208	0.047	0.118	0.066
	M132	0.197	0.046	0.173	0.056	0.249	0.062	0.174	0.088
	M3	0.146	0.028	0.102	0.040	0.207	0.046	0.109	0.061
	M460	0.147	0.042	0.114	0.053	0.218	0.073	0.146	0.098
	M6	0.147	0.041	0.122	0.053	0.217	0.078	0.138	0.098
Order = 4	M0	0.196	0.041	0.217	0.049	0.245	0.064	0.249	0.093
	M132	0.290	0.069	0.308	0.080	0.327	0.109	0.339	0.141
	M3	0.196	0.039	0.115	0.048	0.245	0.069	0.160	0.091
	M460	0.198	0.065	0.433	0.079	0.281	0.122	0.485	0.183
	M6	0.198	0.062	0.204	0.081	0.279	0.116	0.278	0.148
Order = 5	M0	0.242	0.055	0.178	0.064	0.277	0.088	0.200	0.120
	M132	0.341	0.101	0.654	0.116	0.392	0.205	0.646	0.222
	M3	0.196	0.053	0.241	0.066	0.268	0.093	0.264	0.138
	M460	0.478	0.094	0.537	0.111	0.592	0.190	0.580	0.264
	M6	0.382	0.081	0.377	0.096	0.509	0.160	0.427	0.246
Order = 6	M0	0.290	0.070	0.403	0.082	0.324	0.108	0.437	0.148
	M132	0.440	0.141	0.990	0.136	0.492	0.458	1.033	0.297
	M3	0.199	0.068	0.246	0.071	0.281	0.121	0.299	0.164
	M460	0.572	0.128	1.209	0.131	0.695	0.263	1.339	0.320
	M6	0.385	0.109	0.459	0.112	0.547	0.213	0.652	0.292

(a) Quadrangles, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 2	M0	0.144	0.017	0.094	0.029	0.204	0.031	0.111	0.048
	M132	0.195	0.027	0.127	0.033	0.245	0.036	0.142	0.052
	M3	0.144	0.019	0.075	0.025	0.207	0.030	0.099	0.039
	M460	0.145	0.023	0.115	0.030	0.209	0.043	0.145	0.063
	M6	0.145	0.026	0.079	0.033	0.210	0.045	0.106	0.062
Order = 3	M0	0.145	0.024	0.102	0.031	0.207	0.041	0.110	0.055
	M132	0.197	0.046	0.164	0.052	0.248	0.062	0.174	0.089
	M3	0.145	0.028	0.094	0.036	0.207	0.046	0.110	0.061
	M460	0.147	0.042	0.112	0.058	0.218	0.079	0.139	0.099
	M6	0.147	0.042	0.122	0.052	0.219	0.078	0.137	0.102
Order = 4	M0	0.196	0.031	0.224	0.039	0.244	0.053	0.235	0.073
	M132	0.290	0.069	0.308	0.080	0.327	0.108	0.342	0.136
	M3	0.196	0.039	0.120	0.048	0.245	0.069	0.152	0.091
	M460	0.199	0.065	0.429	0.078	0.281	0.121	0.483	0.181
	M6	0.198	0.062	0.205	0.081	0.279	0.116	0.278	0.149
Order = 5	M0	0.242	0.038	0.177	0.052	0.277	0.061	0.200	0.095
	M132	0.340	0.099	0.589	0.106	0.392	0.203	0.588	0.199
	M3	0.196	0.053	0.220	0.058	0.268	0.093	0.242	0.125
	M460	0.477	0.095	0.490	0.101	0.591	0.190	0.531	0.238
	M6	0.382	0.081	0.345	0.089	0.507	0.160	0.389	0.224
Order = 6	M0	0.290	0.045	0.406	0.055	0.324	0.077	0.431	0.113
	M132	0.440	0.138	0.988	0.136	0.492	0.444	1.039	0.294
	M3	0.199	0.068	0.247	0.071	0.282	0.122	0.300	0.164
	M460	0.572	0.129	1.209	0.132	0.695	0.262	1.340	0.319
	M6	0.385	0.109	0.459	0.116	0.547	0.214	0.583	0.294

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Table D.4: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for quadrilateral element matrices in single precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.146	0.028	0.070	0.038	0.211	0.052	0.081	0.068
	M132	0.196	0.032	0.121	0.038	0.249	0.040	0.118	0.057
	M3	0.196	0.032	0.118	0.038	0.249	0.040	0.126	0.056
	M460	0.146	0.027	0.074	0.034	0.211	0.052	0.090	0.070
	M6	0.197	0.042	0.134	0.051	0.245	0.070	0.144	0.102
Order = 2	M0	0.199	0.073	0.434	0.078	0.286	0.131	0.429	0.196
	M132	0.389	0.101	0.624	0.106	0.424	0.250	0.612	0.186
	M3	0.290	0.073	0.327	0.086	0.326	0.138	0.392	0.142
	M460	0.385	0.094	0.626	0.102	0.507	0.183	0.649	0.268
	M6	0.575	0.118	0.882	0.126	0.679	0.239	0.905	0.286
Order = 3	M0	0.574	0.142	0.882	0.144	0.675	0.387	0.920	0.443
	M132	0.689	0.337	1.771	0.405	0.742	0.763	1.980	0.846
	M3	0.394	0.142	0.887	0.146	0.453	0.488	0.915	0.559
	M460	0.897	0.226	1.737	0.237	1.237	0.720	1.821	0.750
	M6	1.060	0.252	2.566	0.276	1.398	1.226	2.621	1.429
Order = 4	M0	1.131	0.239	4.117	0.318	1.506	1.066	4.241	1.124
	M132	2.544	1.221	9.755	0.983	2.646	1.397	9.872	1.632
	M3	1.061	0.222	3.020	0.331	1.144	0.846	3.147	1.014
	M460	2.419	0.426	9.852	0.514	2.823	2.251	10.17	2.539
	M6	2.799	0.474	8.877	0.531	3.056	2.130	9.239	2.448
Order = 5	M0	2.237	0.359	6.445	0.521	2.430	1.625	6.431	1.701
	M132	5.465	6.851	19.22	2.439	5.611	2.805	18.66	3.348
	M3	2.312	0.417	6.459	0.508	2.440	1.450	6.366	1.688
	M460	5.735	1.339	19.44	1.295	6.585	4.338	19.13	4.559
	M6	5.510	0.935	19.44	0.862	6.577	3.820	19.12	4.167
Order = 6	M0	4.209	2.846	21.00	2.044	4.466	3.533	21.41	3.140
	M132	13.33	13.67	72.01	4.825	13.75	7.002	72.39	6.805
	M3	4.202	1.230	15.72	0.870	4.890	2.488	16.09	2.683
	M460	12.71	3.744	71.14	3.806	14.21	7.542	72.25	8.727
	M6	10.80	1.397	46.89	1.623	12.34	5.727	48.08	6.642

(c) Hexahedra, Gauss-Legendre Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 2	M0	0.199	0.071	0.435	0.088	0.286	0.128	0.485	0.195
	M132	0.390	0.102	0.631	0.116	0.424	0.248	0.700	0.188
	M3	0.290	0.073	0.330	0.090	0.329	0.139	0.397	0.157
	M460	0.385	0.094	0.626	0.101	0.508	0.201	0.692	0.315
	M6	0.576	0.118	0.856	0.121	0.681	0.237	0.906	0.278
Order = 3	M0	0.574	0.135	0.882	0.137	0.676	0.360	0.923	0.366
	M132	0.690	0.296	1.767	0.343	0.742	0.620	1.760	0.711
	M3	0.394	0.137	0.887	0.145	0.453	0.491	0.906	0.549
	M460	0.890	0.224	1.737	0.240	1.237	0.714	1.820	0.744
	M6	1.060	0.252	2.566	0.276	1.398	1.219	2.613	1.199
Order = 4	M0	1.133	0.217	4.124	0.226	1.509	0.427	4.251	0.619
	M132	2.545	1.328	9.835	0.945	2.654	1.347	9.879	1.609
	M3	1.133	0.243	3.020	0.331	1.147	0.849	3.152	1.014
	M460	2.451	0.448	9.858	0.514	2.848	2.232	10.16	2.524
	M6	2.735	0.426	8.886	0.530	3.058	2.130	9.226	2.449
Order = 5	M0	2.309	0.306	6.529	0.417	2.694	0.724	6.355	1.510
	M132	5.240	5.936	19.18	2.190	5.625	2.696	18.67	3.250
	M3	2.246	0.434	6.461	0.504	2.743	1.485	6.371	1.701
	M460	5.559	1.258	19.46	1.221	6.556	4.231	19.13	4.700
	M6	5.640	0.936	19.43	0.866	6.424	3.874	19.13	4.163
Order = 6	M0	4.100	0.699	20.99	0.581	4.436	1.803	21.38	2.042
	M132	13.25	13.47	71.94	4.909	13.83	6.566	72.43	6.991
	M3	4.434	1.205	15.73	0.973	4.860	2.508	16.10	2.665
	M460	12.63	3.114	71.12	3.550	14.10	7.177	72.23	8.152
	M6	10.87	1.345	46.90	1.633	12.37	5.722	48.07	6.718

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table D.4: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for hexahedral element matrices in single precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.038	0.009	0.044	0.012	0.076	0.016	0.064	0.018
	M132	0.039	0.010	0.037	0.012	0.064	0.013	0.035	0.015
	M3	0.039	0.011	0.035	0.014	0.064	0.014	0.041	0.017
	M460	0.077	0.009	0.040	0.012	0.076	0.016	0.077	0.020
	M6	0.042	0.012	0.074	0.017	0.080	0.020	0.104	0.025
Order = 2	M0	0.144	0.014	0.064	0.018	0.202	0.025	0.091	0.038
	M132	0.060	0.019	0.072	0.024	0.095	0.025	0.106	0.032
	M3	0.060	0.015	0.097	0.020	0.094	0.022	0.142	0.029
	M460	0.144	0.016	0.056	0.024	0.202	0.030	0.079	0.038
	M6	0.144	0.018	0.090	0.024	0.204	0.033	0.106	0.047
Order = 3	M0	0.144	0.019	0.083	0.029	0.206	0.034	0.091	0.043
	M132	0.195	0.030	0.103	0.040	0.245	0.043	0.153	0.055
	M3	0.144	0.021	0.081	0.030	0.207	0.033	0.120	0.041
	M460	0.145	0.026	0.081	0.034	0.209	0.048	0.105	0.066
	M6	0.145	0.028	0.075	0.038	0.212	0.054	0.100	0.066
Order = 4	M0	0.145	0.026	0.142	0.040	0.207	0.044	0.198	0.057
	M132	0.196	0.045	0.202	0.058	0.246	0.066	0.260	0.089
	M3	0.145	0.026	0.144	0.035	0.206	0.044	0.203	0.059
	M460	0.147	0.039	0.153	0.048	0.216	0.071	0.197	0.093
	M6	0.147	0.039	0.145	0.047	0.216	0.078	0.197	0.093
Order = 5	M0	0.196	0.037	0.200	0.050	0.245	0.062	0.223	0.074
	M132	0.243	0.066	0.248	0.089	0.278	0.097	0.276	0.130
	M3	0.195	0.034	0.125	0.050	0.242	0.059	0.156	0.076
	M460	0.198	0.056	0.354	0.069	0.273	0.115	0.424	0.130
	M6	0.196	0.053	0.251	0.061	0.273	0.106	0.316	0.124
Order = 6	M0	0.196	0.048	0.132	0.063	0.244	0.078	0.166	0.110
	M132	0.291	0.123	0.311	0.141	0.326	0.159	0.320	0.239
	M3	0.196	0.044	0.206	0.058	0.244	0.084	0.234	0.100
	M460	0.200	0.106	0.296	0.106	0.288	0.162	0.333	0.229
	M6	0.199	0.067	0.368	0.077	0.287	0.140	0.424	0.205

(e) Triangles, Williams-Shunn Method

Matrix		$\beta = 0$				$\beta \neq 0$			
		CU	GCU	CL	GCL	CU	GCU	CL	GCL
Order = 1	M0	0.143	0.016	0.048	0.018	0.199	0.028	0.058	0.036
	M132	0.057	0.017	0.063	0.022	0.087	0.021	0.065	0.031
	M3	0.057	0.017	0.063	0.025	0.087	0.022	0.065	0.028
	M460	0.199	0.016	0.046	0.018	0.198	0.028	0.051	0.037
	M6	0.144	0.021	0.080	0.035	0.205	0.036	0.091	0.051
Order = 2	M0	0.146	0.030	0.082	0.037	0.212	0.055	0.110	0.072
	M132	0.196	0.040	0.194	0.053	0.249	0.053	0.224	0.074
	M3	0.196	0.035	0.126	0.042	0.249	0.048	0.162	0.067
	M460	0.146	0.035	0.105	0.045	0.215	0.066	0.163	0.089
	M6	0.196	0.047	0.140	0.057	0.245	0.085	0.196	0.124
Order = 3	M0	0.197	0.053	0.215	0.064	0.272	0.109	0.241	0.157
	M132	0.293	0.091	0.275	0.121	0.329	0.206	0.282	0.193
	M3	0.243	0.061	0.190	0.080	0.278	0.092	0.205	0.125
	M460	0.199	0.072	0.240	0.086	0.290	0.152	0.278	0.221
	M6	0.246	0.095	0.433	0.107	0.327	0.191	0.469	0.246
Order = 4	M0	0.246	0.110	0.554	0.131	0.325	0.188	0.613	0.264
	M132	0.439	0.372	1.152	0.292	0.485	0.490	1.219	0.451
	M3	0.292	0.140	0.460	0.160	0.349	0.332	0.579	0.280
	M460	0.480	0.192	0.925	0.217	0.624	0.359	1.058	0.381
	M6	0.576	0.242	0.989	0.192	0.703	0.621	1.091	0.526
Order = 5	M0	0.575	0.524	0.733	0.364	0.677	0.695	0.774	0.533
	M132	0.644	1.817	1.392	1.555	0.692	2.022	1.382	1.810
	M3	0.394	0.540	0.727	0.434	0.448	0.652	0.723	0.501
	M460	0.733	0.965	1.469	0.669	1.166	1.364	1.541	1.001
	M6	0.934	1.411	2.163	0.619	1.319	1.156	2.205	1.030
Order = 6	M0	0.773	1.001	1.467	0.785	0.879	1.213	1.490	1.008
	M132	1.767	3.946	3.137	3.524	1.851	4.098	3.088	3.886
	M3	0.868	1.819	1.414	1.561	0.853	1.075	1.430	0.988
	M460	1.376	4.313	3.072	4.341	1.833	2.486	3.150	2.447
	M6	1.551	2.735	3.962	2.281	1.768	1.816	3.989	3.340

(f) Tetrahedra, Shunn-Ham Method

Table D.4: Benchmarking results for GiMMiK CUDA (GCU) and OpenCL (GCL) kernels, cuBLAS (CU) and clBLAS(CL) for triangular and tetrahedral element matrices in single precision on GTX 780 Ti. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$	
		CL	GCL
Order = 1	M0	0.043	0.027
	M132	0.038	0.028
	M3	0.034	0.028
	M460	0.043	0.030
	M6	0.050	0.035
Order = 2	M0	0.080	0.045
	M132	0.077	0.059
	M3	0.064	0.046
	M460	0.114	0.059
	M6	0.099	0.063
Order = 3	M0	0.074	0.069
	M132	0.111	0.124
	M3	0.073	0.067
	M460	0.126	0.109
	M6	0.127	0.099
Order = 4	M0	0.189	0.105
	M132	0.263	0.231
	M3	0.138	0.098
	M460	0.357	0.198
	M6	0.261	0.142
Order = 5	M0	0.203	0.160
	M132	0.526	0.537
	M3	0.225	0.130
	M460	0.492	0.422
	M6	0.370	0.199
Order = 6	M0	0.306	0.302
	M132	0.914	0.710
	M3	0.328	0.176
	M460	0.973	0.627
	M6	0.567	0.253

(a) Quadrangles, Gauss-Legendre Method

Matrix		$\beta = 0$	
		CL	GCL
Order = 2	M0	0.081	0.045
	M132	0.075	0.055
	M3	0.061	0.045
	M460	0.113	0.057
	M6	0.097	0.064
Order = 3	M0	0.073	0.058
	M132	0.113	0.116
	M3	0.073	0.068
	M460	0.124	0.108
	M6	0.123	0.100
Order = 4	M0	0.189	0.073
	M132	0.263	0.224
	M3	0.137	0.096
	M460	0.357	0.191
	M6	0.264	0.140
Order = 5	M0	0.203	0.089
	M132	0.527	0.517
	M3	0.227	0.133
	M460	0.489	0.412
	M6	0.373	0.195
Order = 6	M0	0.301	0.101
	M132	0.902	0.702
	M3	0.327	0.175
	M460	0.973	0.594
	M6	0.567	0.248

(b) Quadrangles, Gauss-Legendre-Lobatto Method

Matrix		$\beta = 0$	
		CL	GCL
Order = 1	M0	0.087	0.064
	M132	0.085	0.065
	M3	0.089	0.065
	M460	0.086	0.064
	M6	0.155	0.097
Order = 2	M0	0.380	0.205
	M132	0.444	0.373
	M3	0.283	0.186
	M460	0.551	0.250
	M6	0.797	0.274
Order = 3	M0	0.933	0.495
	M132	1.727	2.036
	M3	0.900	0.474
	M460	1.835	2.435
	M6	2.620	0.583
Order = 4	M0	3.040	4.900
	M132	6.858	18.21
	M3	2.651	1.685
	M460	7.230	14.84
	M6	7.860	1.107
Order = 5	M0	6.478	14.47
	M132	9.312	41.576
	M3	6.489	7.403
	M460	9.249	38.590
	M6	8.900	4.775
Order = 6	M0	4.917	23.436
	M132	1.163	87.188
	M3	3.951	11.913
	M460	1.473	70.084
	M6	0.528	7.353

(c) Hexahedra, Gauss-Legendre Method

Matrix		$\beta = 0$	
		CL	GCL
Order = 2	M0	0.379	0.168
	M132	0.448	0.368
	M3	0.283	0.186
	M460	0.548	0.244
	M6	0.801	0.273
Order = 3	M0	0.929	0.287
	M132	1.701	1.999
	M3	0.901	0.476
	M460	1.837	1.729
	M6	2.619	0.574
Order = 4	M0	3.034	0.484
	M132	6.853	16.434
	M3	2.648	1.598
	M460	7.237	12.508
	M6	7.997	1.082
Order = 5	M0	6.470	0.728
	M132	19.320	40.442
	M3	6.476	6.981
	M460	19.225	32.761
	M6	18.900	4.456
Order = 6	M0	15.092	1.056
	M132	50.840	77.815
	M3	13.947	15.039
	M460	51.496	63.601
	M6	40.606	8.995

(d) Hexahedra, Gauss-Legendre-Lobatto Method

Table D.5: Benchmarking results for GiMMiK OpenCL (GCL) kernels and clBLAS(CL) for quadrilateral and hexahedral element matrices in double precision on Tesla K40c. Reported values are averages of 30 runs reproducible within 2% in [ms].

Matrix		$\beta = 0$	
		CL	GCL
Order = 1	M0	0.036	0.021
	M132	0.034	0.024
	M3	0.034	0.022
	M460	0.310	0.013
	M6	0.042	0.027
Order = 2	M0	0.048	0.035
	M132	0.066	0.048
	M3	0.060	0.041
	M460	0.053	0.043
	M6	0.080	0.052
Order = 3	M0	0.065	0.079
	M132	0.081	0.128
	M3	0.064	0.072
	M460	0.092	0.125
	M6	0.102	0.096
Order = 4	M0	0.101	0.127
	M132	0.113	0.276
	M3	0.128	0.101
	M460	0.154	0.256
	M6	0.153	0.228
Order = 5	M0	0.173	0.324
	M132	0.227	0.568
	M3	0.126	0.213
	M460	0.251	0.666
	M6	0.191	0.525
Order = 6	M0	0.171	0.503
	M132	0.293	2.577
	M3	0.179	0.319
	M460	0.336	4.826
	M6	0.413	2.276

(e) Triangles, Williams-Shunn Method

Matrix		$\beta = 0$	
		CL	GCL
Order = 1	M0	0.046	0.035
	M132	0.045	0.035
	M3	0.046	0.037
	M460	0.039	0.026
	M6	0.066	0.052
Order = 2	M0	0.095	0.099
	M132	0.102	0.207
	M3	0.093	0.127
	M460	0.102	0.169
	M6	0.163	0.227
Order = 3	M0	0.203	1.052
	M132	0.302	1.006
	M3	0.218	0.656
	M460	0.273	3.816
	M6	0.435	4.121
Order = 4	M0	0.454	13.280
	M132	0.818	14.955
	M3	0.447	7.870
	M460	0.759	25.123
	M6	1.027	20.207
Order = 5	M0	0.832	39.117
	M132	1.498	66.123
	M3	0.791	22.023
	M460	1.503	83.452
	M6	2.134	57.742
Order = 6	M0	1.364	121.863
	M132	3.252	423.069
	M3	1.526	59.262
	M460	3.096	197.345
	M6	4.017	132.365

(f) Tetrahedra, Shunn-Ham Method

Table D.5: Benchmarking results for GiMMiK OpenCL (GCL) kernels and clBLAS(CL) for triangular and tetrahedral element matrices in double precision on FirePro W9100. Reported values are averages of 30 runs reproducible within 2% in [ms].

Appendix E

Plots of Benchmarking and Profiling Results for GiMMiK Kernels

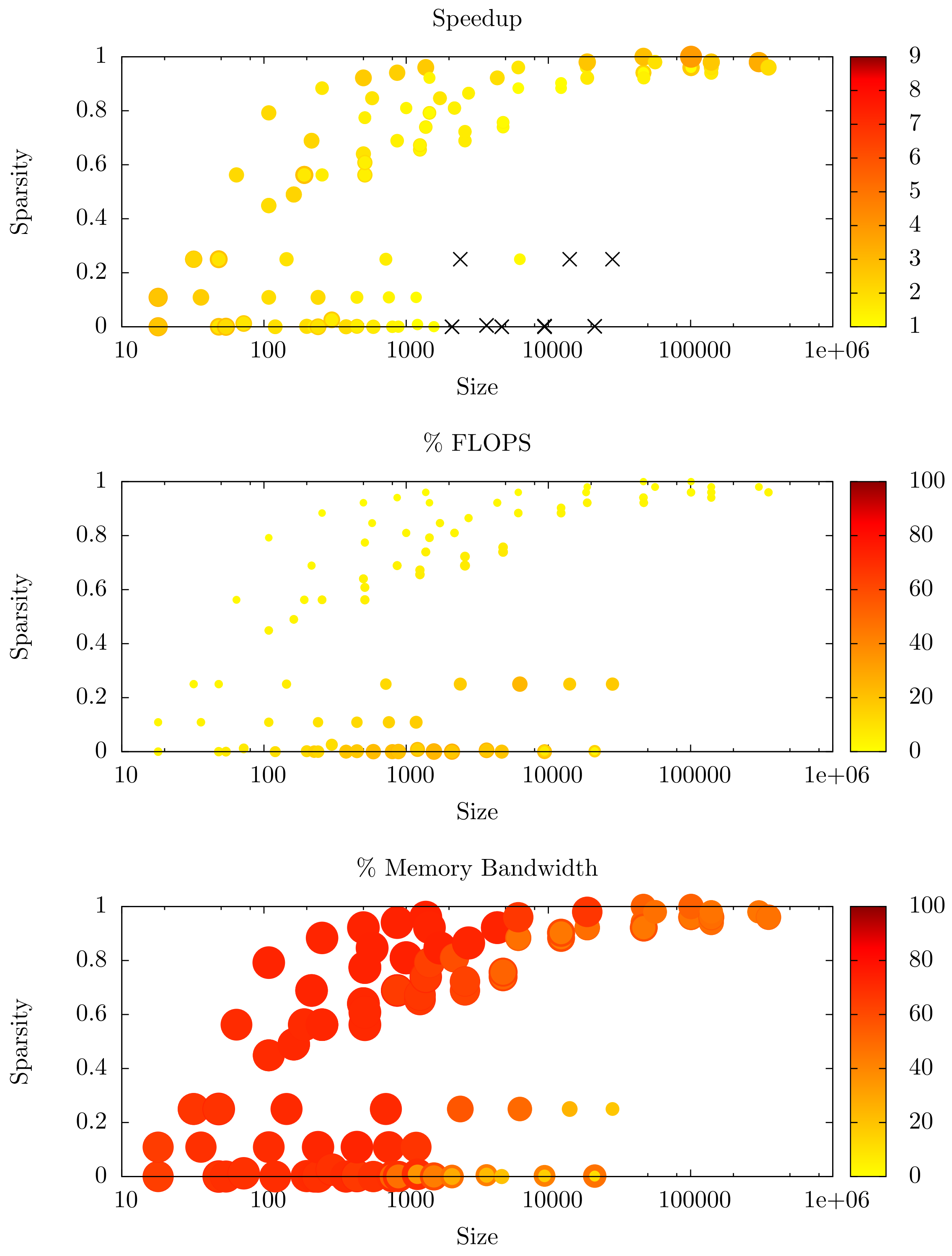


Figure E.1: Plots illustrating the speedup of GiMMiK's CUDA kernels over cuBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision, $\beta \neq 0$ on Tesla K40c.

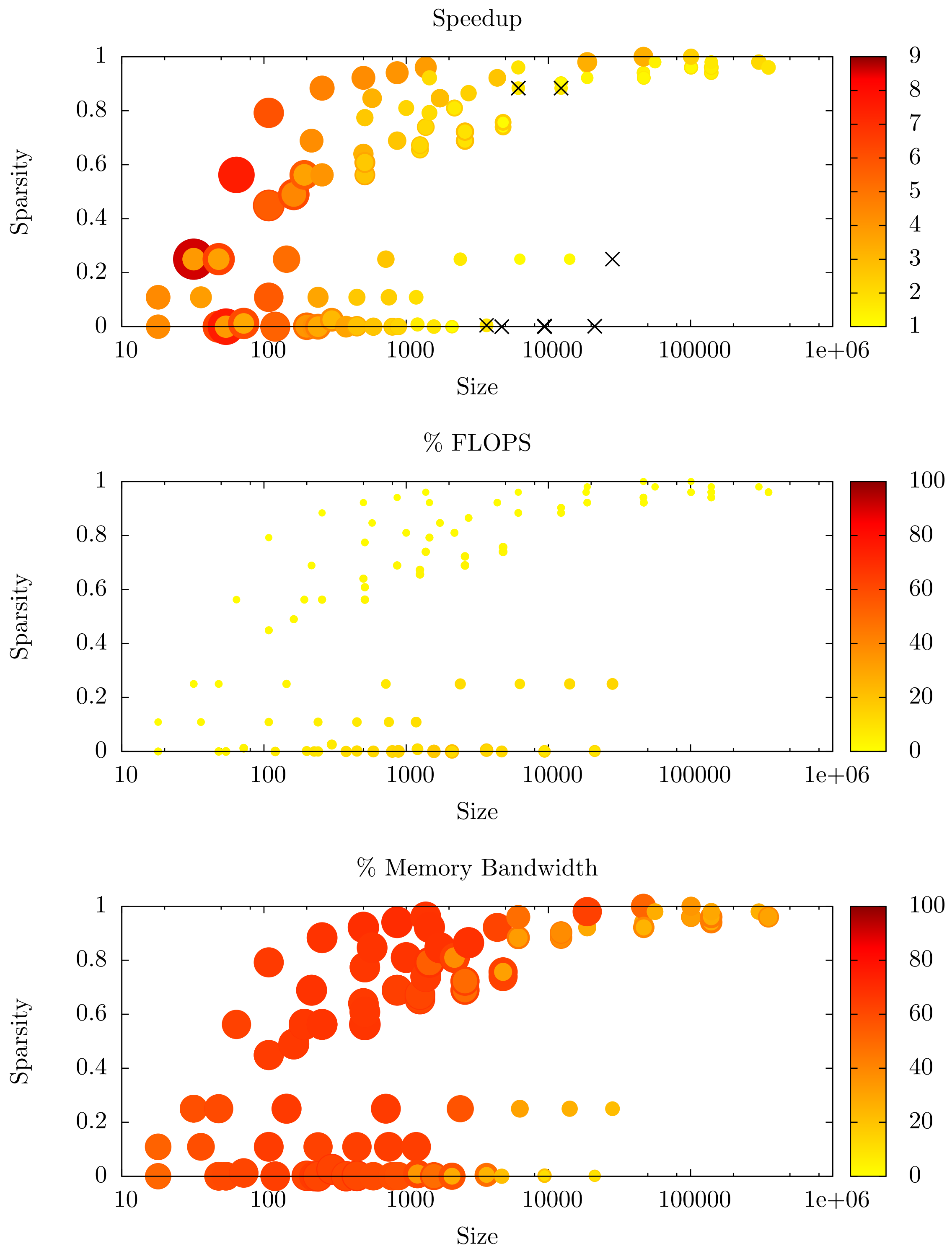


Figure E.2: Plots illustrating the speedup of GiMMiK's CUDA kernels over cuBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision, $\beta \neq 0$ on Tesla K40c.

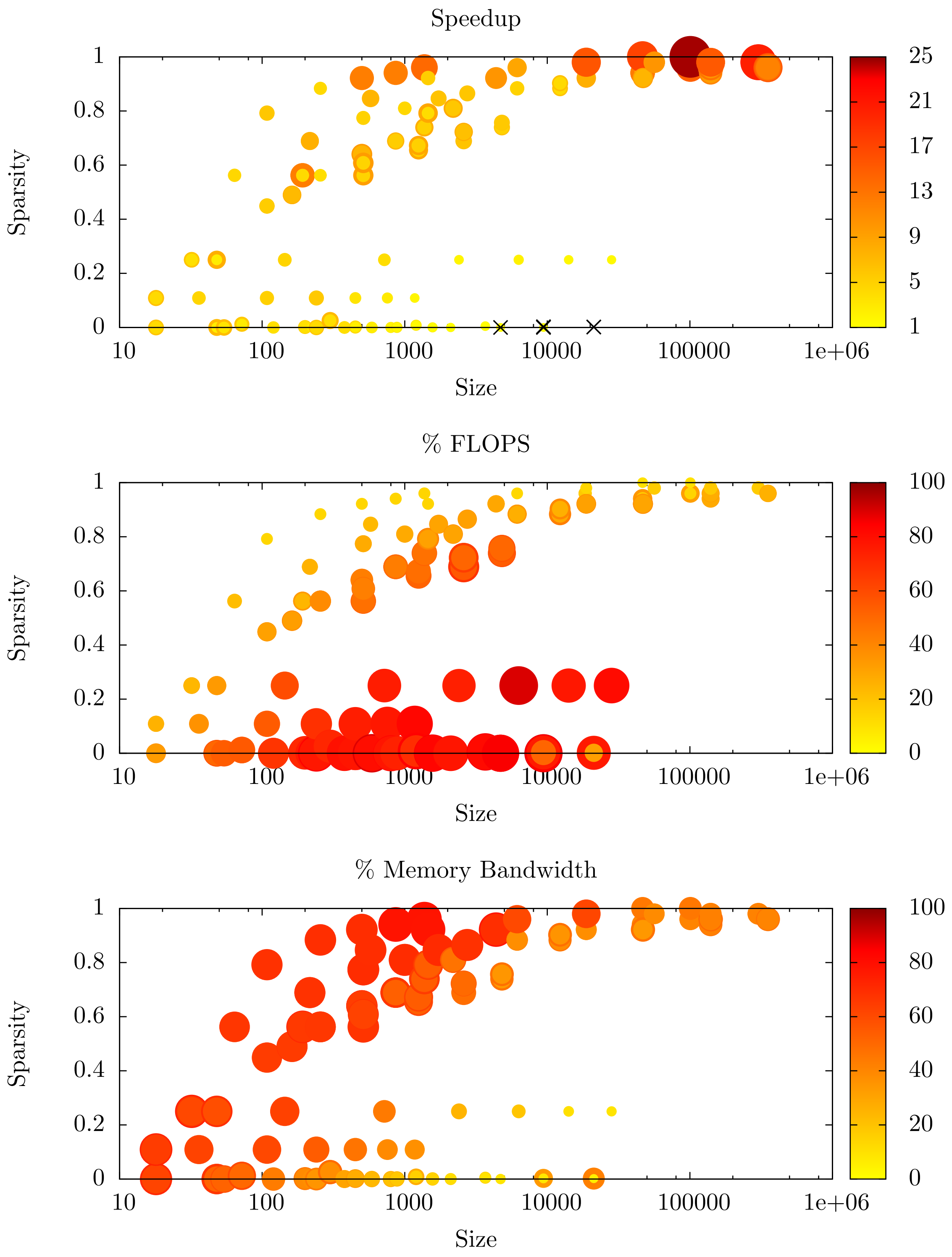


Figure E.3: Plots illustrating the speedup of GiMMiK's CUDA kernels over cuBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision, $\beta \neq 0$ on GTX 780 Ti.

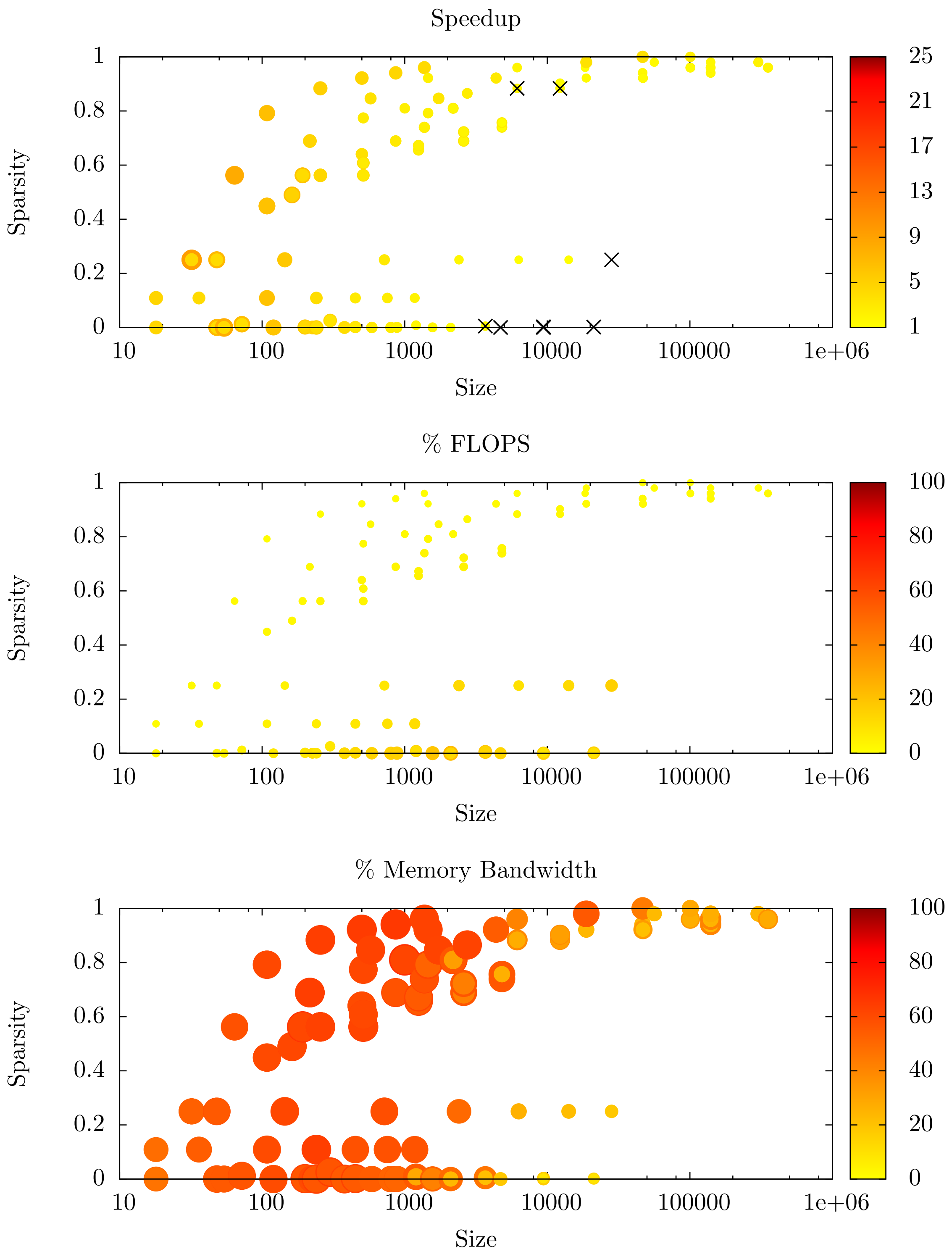


Figure E.4: Plots illustrating the speedup of GiMMiK's CUDA kernels over cuBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision, $\beta \neq 0$ on GTX 780 Ti.

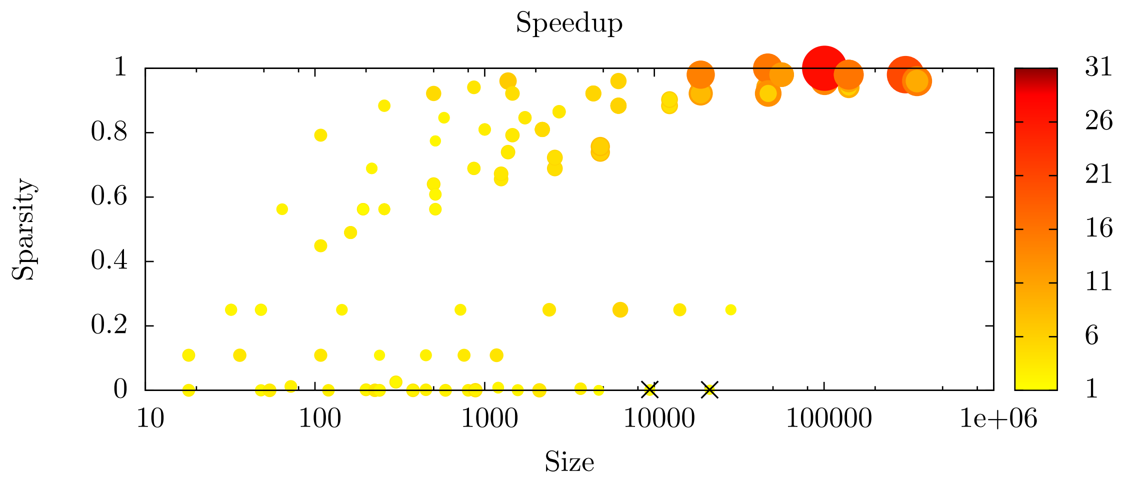


Figure E.5: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision, $\beta = 0$ on Tesla K40c.

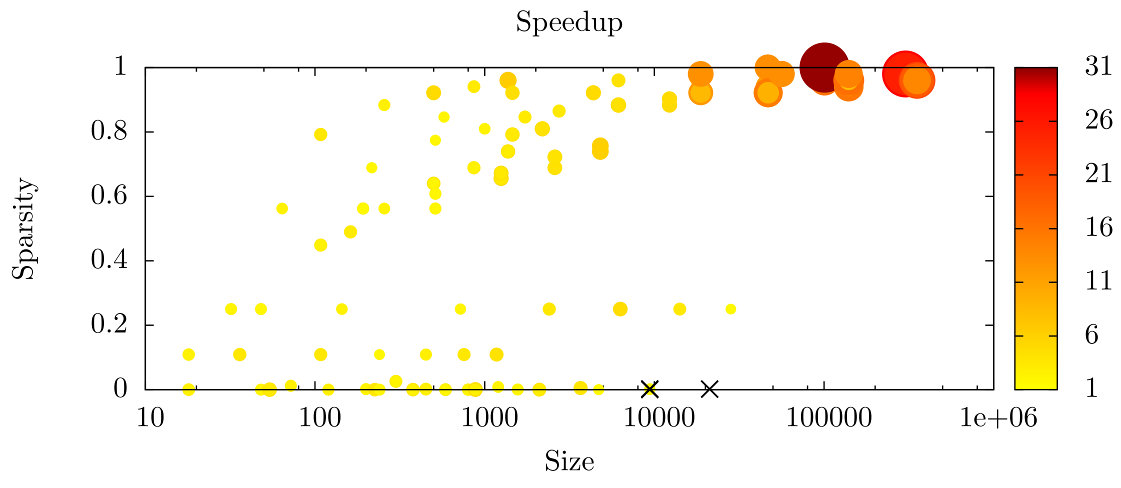


Figure E.6: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision, $\beta = 0$ on Tesla K40c.

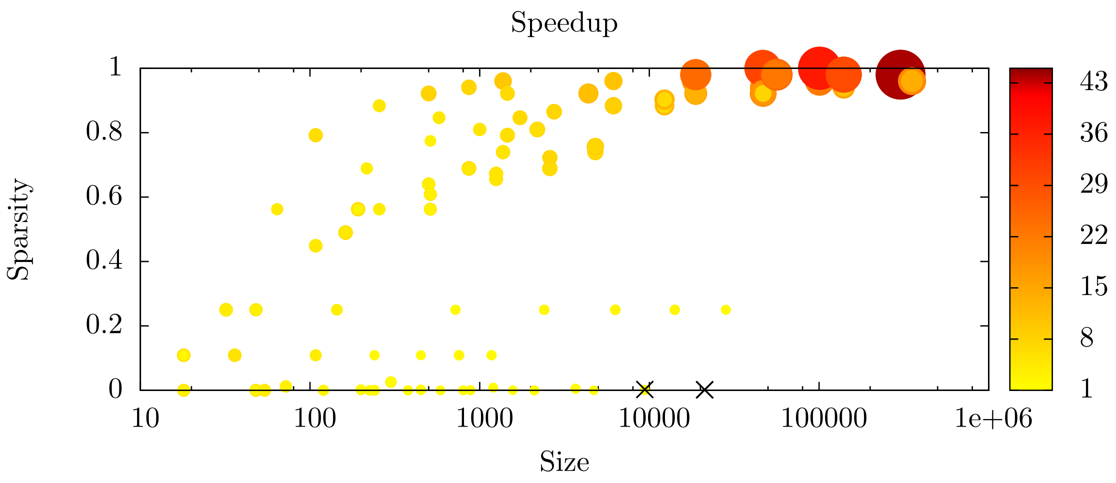


Figure E.7: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision, $\beta = 0$ on GTX 780 Ti.

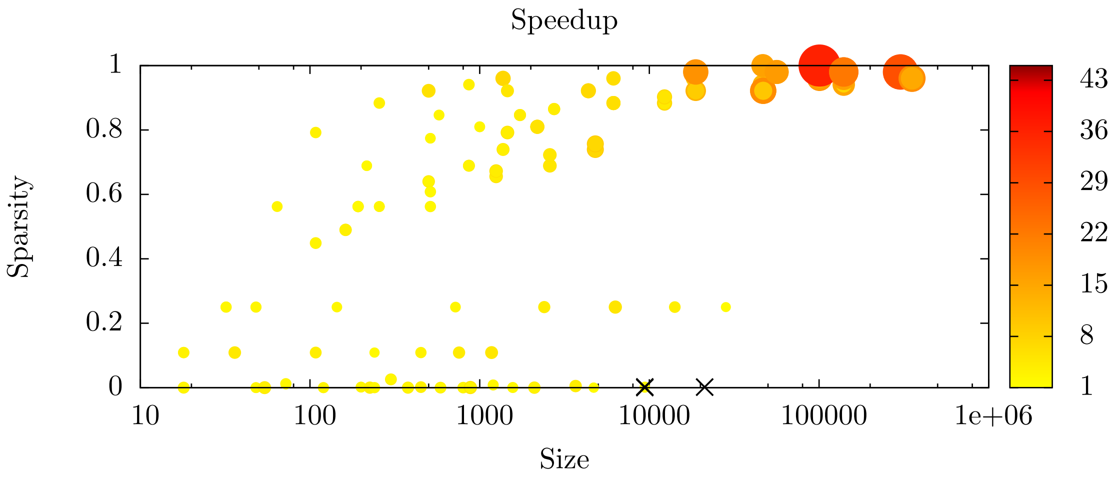


Figure E.8: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision, $\beta = 0$ on GTX 780 Ti.

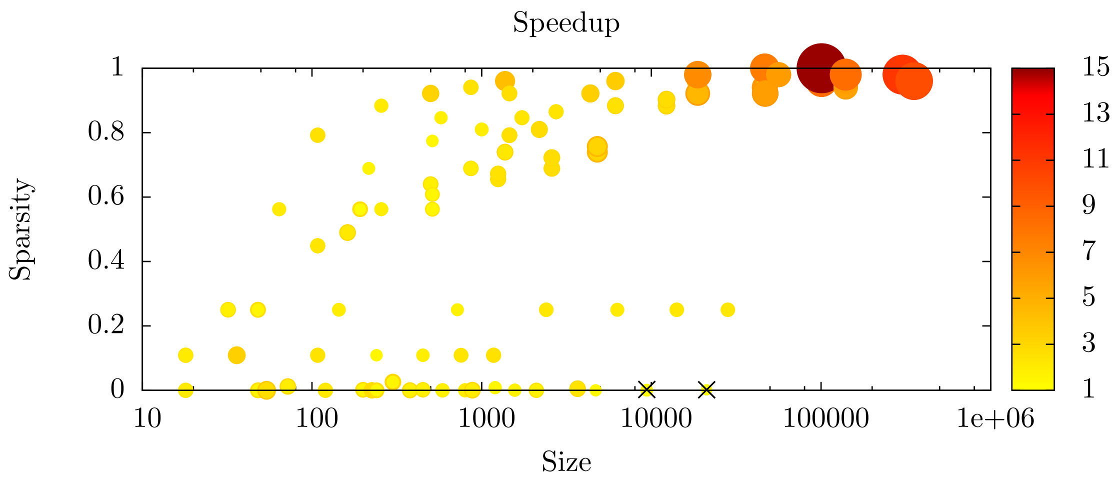


Figure E.9: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision, $\beta \neq 0$ on Tesla K40c.

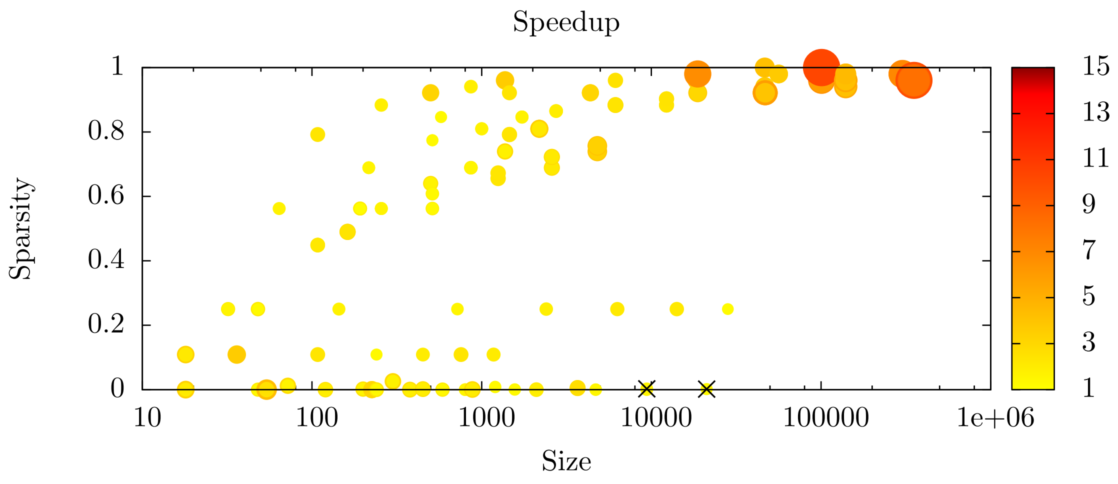


Figure E.10: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision, $\beta \neq 0$ on Tesla K40c.

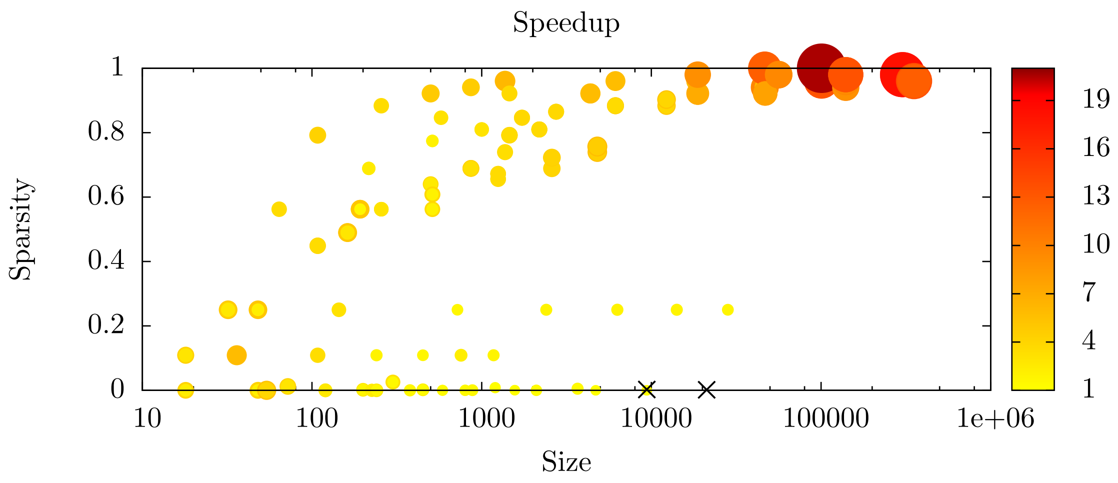


Figure E.11: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for double precision, $\beta \neq 0$ on GTX 780 Ti.

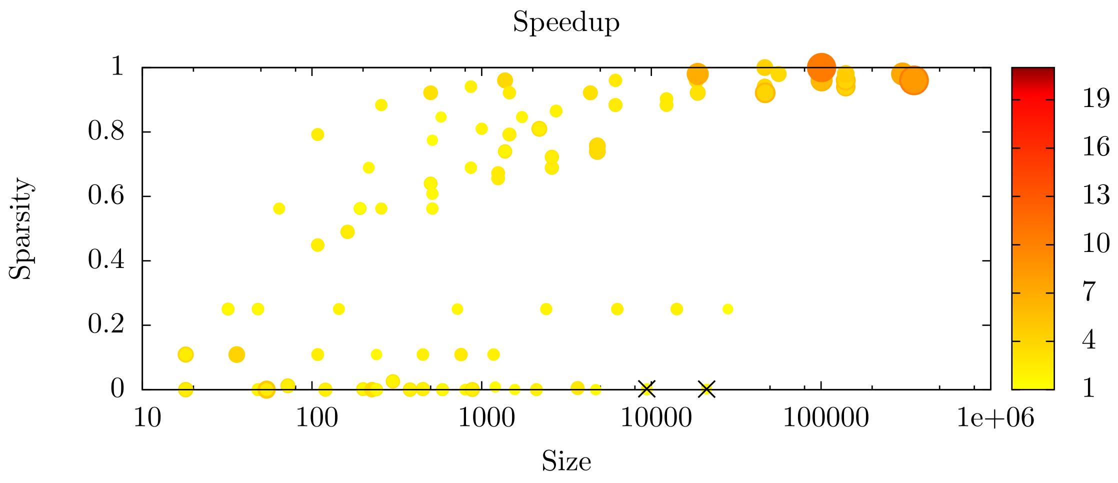


Figure E.12: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision, $\beta \neq 0$ on GTX 780 Ti.

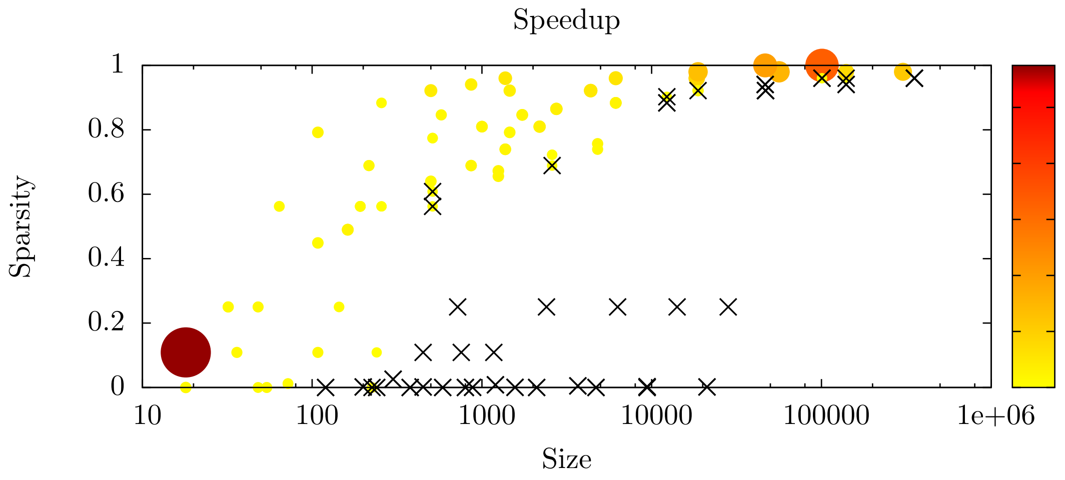


Figure E.13: Plots illustrating the speedup of GiMMiK's OpenCL kernels over clBLAS, the achieved percentage of the peak floating-point rate and the achieved percentage of the peak memory bandwidth. The metric of interest is represented through the size and colour intensity of the data points. Speedups smaller than 1 are denoted with crosses. These plots are for single precision, $\beta = 0$ on FirePro W9100.

Appendix F

Speedups of Individual Matrices Stacked Together to Mimic PyFR

From a practical standpoint of running a PyFR simulation, the most interesting case make matrices for the third order of accuracy. This order provides a good balance between the accuracy and the cost of a simulation. For the sake of brevity only stacked plots for the third order of accuracy are shown in this appendix. The procedure used to generate this data is described in Section 5.4. Table F.1 gives the results of aggregating the benchmarking results for individual matrices to mimic the matrix multiplication steps executed by PyFR.

Order		Tesla K40c						GTX 780 Ti					
		double			single			double			single		
		MIN	CU	GIM	MIN	CU	GIM	MIN	CU	GIM	MIN	CU	GIM
Quad GL	1	0.12222	1.04084	0.39413	0.06111	2.07011	0.21131	0.12190	1.76332	0.30432	0.05238	1.75416	0.16536
	2	0.23333	1.87941	0.73505	0.11667	2.42213	0.37752	0.46286	4.93559	0.56352	0.10000	2.04934	0.28987
	3	0.37778	2.11323	1.17074	0.18889	2.45865	0.60493	1.21905	5.66471	0.92738	0.16190	2.07822	0.46263
	4	0.55556	4.08358	1.74141	0.27778	3.29805	0.91875	2.61905	15.17051	1.38804	0.23810	2.77941	0.68706
	5	0.82918	5.30554	2.42938	0.38333	4.47383	1.26205	4.93714	20.09307	1.97624	0.32857	3.77489	0.94502
	6	1.29127	6.57193	3.23086	0.52526	5.27793	1.66864	8.49333	26.50802	2.59411	0.44889	4.44918	1.23705
Quad GLL	2	0.23333	1.87852	0.71180	0.11667	2.41813	0.36722	0.46286	4.93944	0.54568	0.10000	2.04885	0.28185
	3	0.37778	2.11155	1.08979	0.18889	2.45853	0.56795	1.21905	5.66213	0.82716	0.16190	2.07831	0.43098
	4	0.55556	4.09016	1.51782	0.27778	3.29725	0.79472	2.61905	15.30378	1.16253	0.23810	2.78030	0.60205
	5	0.82918	5.30703	2.00781	0.38333	4.47612	1.04532	4.93714	19.90382	1.50684	0.32857	3.76898	0.78801
	6	1.29127	6.57023	2.62584	0.52526	5.27516	1.35373	8.49333	26.66911	2.10155	0.44889	4.45032	1.01100
Hex GL	1	0.37778	3.82531	1.54707	0.18889	3.73195	0.80525	0.91429	9.52045	1.16760	0.16190	3.17015	0.61017
	2	1.17425	7.68053	4.42025	0.53321	5.87925	2.30543	7.63714	28.98212	3.24751	0.45643	4.96746	1.73364
	3	5.15580	20.96115	10.21770	1.71860	14.90109	5.93266	35.10857	88.20773	7.29273	1.46286	12.61362	4.54491
	4	17.04545	57.94878	18.49778	5.68182	32.84768	11.09661	116.07143	248.34311	13.37428	4.83631	27.32805	8.22743
	5	45.67720	131.41853	95.78757	15.22573	69.27776	26.66149	311.04000	610.77370	73.98021	12.96000	56.00637	19.20344
	6	105.77832	266.65506	169.60655	35.25944	135.09674	91.57254	720.30000	1,273.75290	149.57296	30.01250	110.62222	71.16811
Hex GLL	2	1.17425	7.67273	4.25729	0.53321	5.88371	2.25339	7.63714	28.87332	3.16821	0.45643	4.96988	1.70119
	3	5.15580	20.94030	9.79230	1.71860	14.90446	5.71507	35.10857	84.36043	6.78652	1.46286	12.61070	4.38448
	4	17.04545	57.94878	18.49778	5.68182	32.84768	11.09661	116.07143	248.34311	13.37428	4.83631	27.32805	8.22743
	5	45.67720	57.94878	18.49778	15.22573	32.84768	11.09661	311.04000	248.34311	13.37428	12.96000	27.32805	8.22743
	6	105.77832	266.64810	59.03600	35.25944	135.19557	48.04014	720.30000	1,272.98200	44.12210	30.01250	108.72125	36.00467
Tri WS	1	0.09167	0.96517	0.29401	0.04583	0.67545	0.16764	0.08143	1.51382	0.24233	0.03929	0.60024	0.13092
	2	0.16250	1.23234	0.51344	0.08125	2.11870	0.26760	0.22286	2.48460	0.51847	0.06964	1.79470	0.21324
	3	0.25000	1.88676	0.78864	0.12500	2.42358	0.41261	0.53333	4.93532	1.05211	0.10714	2.05383	0.31801
	4	0.35417	2.09053	1.12323	0.17708	2.44603	0.58209	1.07143	5.66130	1.97071	0.15179	2.07171	0.44119
	5	0.47500	3.62123	1.55895	0.23750	3.22285	0.80526	1.92000	12.45481	3.40634	0.20357	2.71772	0.61769
	6	0.61250	4.16601	2.18110	0.30625	3.31062	1.12791	3.17333	15.05984	5.61150	0.26250	2.78907	0.88783
Tet SH	1	0.16667	1.15492	0.53302	0.08333	2.10136	0.28015	0.20571	1.94448	0.39264	0.07143	1.83832	0.23002
	2	0.37500	3.04855	1.20274	0.18750	2.53801	0.62169	1.08571	9.41494	1.75168	0.16071	2.14690	0.47424
	3	0.80672	4.76654	2.96717	0.38889	4.10586	1.43331	4.19048	16.57924	7.48189	0.33333	3.45557	1.08318
	4	1.54196	6.30111	6.00279	0.60519	4.95625	3.17512	10.50000	24.45646	17.57326	0.51786	4.18542	2.50998
	5	3.61846	12.99464	24.46414	1.20615	9.83790	11.39173	24.64000	50.18688	38.68229	1.02667	8.31631	9.30700
	6	7.56587	21.08438	58.98790	2.52196	15.38387	25.72484	51.52000	88.29794	85.28484	2.14667	12.72199	20.16244

Table F.1: Benchmarking results for individual matrices have been combined together to mimic the matrix multiplication steps executed by PyFR. MIN is the minimum time required by any dense GEMM based on the peak floating-point rate and memory bandwidth of the device. CU is the matrix-multiplication step performed by cuBLAS GEMM and GIM is the step performed by GiMMiK. Times reported in [ms].

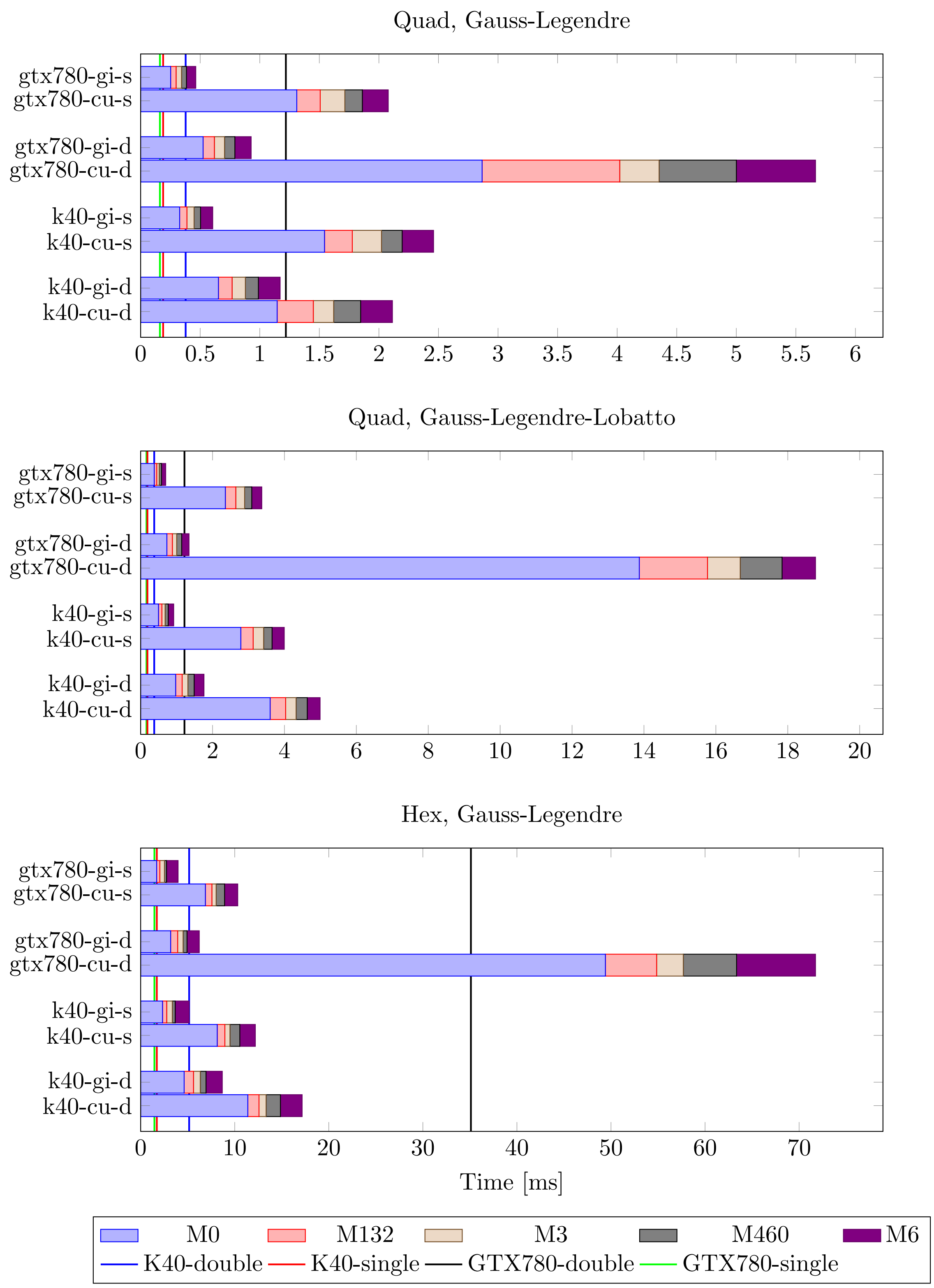


Figure F.1: Combined time for execution of a single time step in PyFR for quadrilateral elements mesh for 3rd order of accuracy and the theoretical limits imposed by the devices.

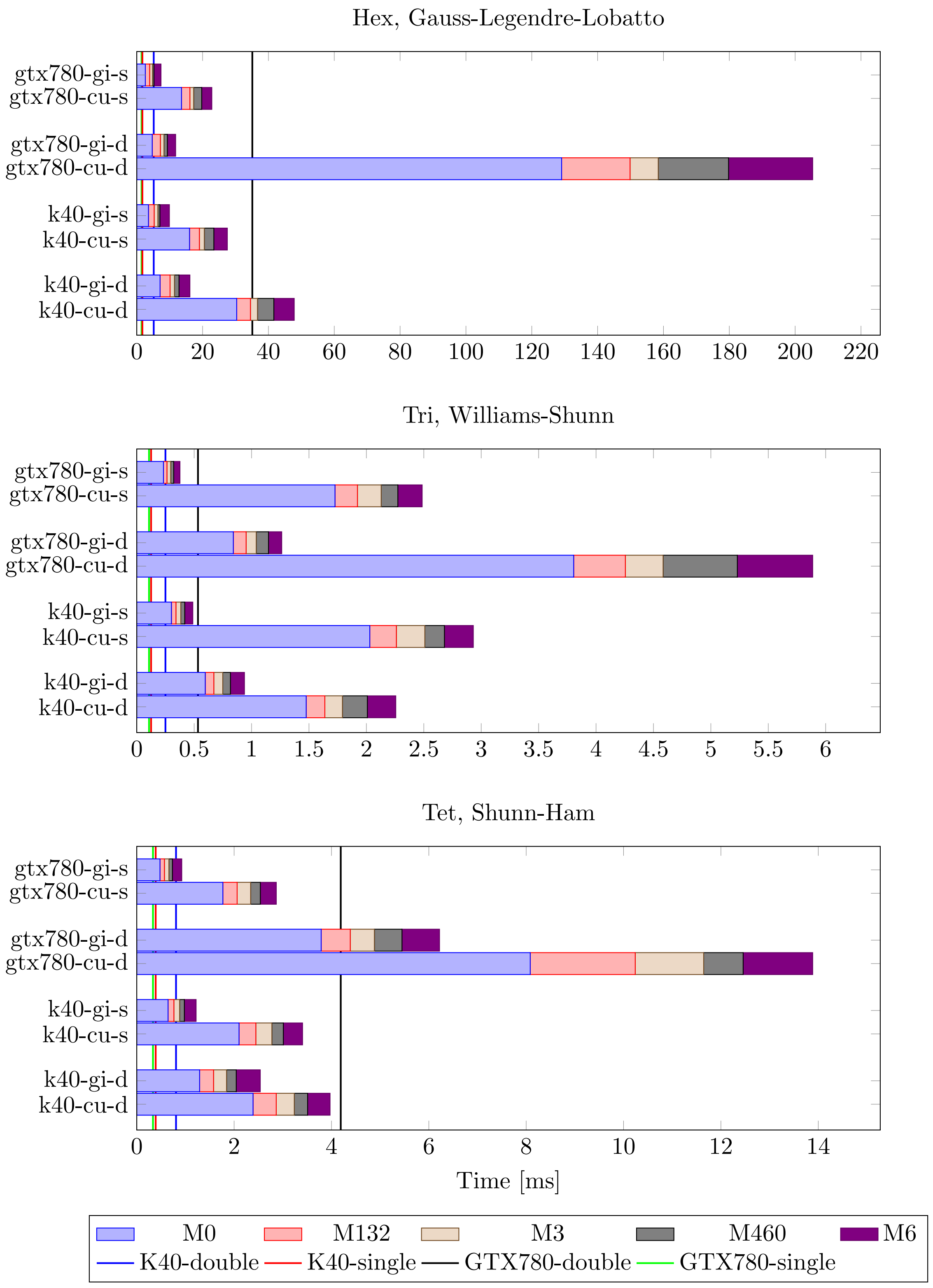
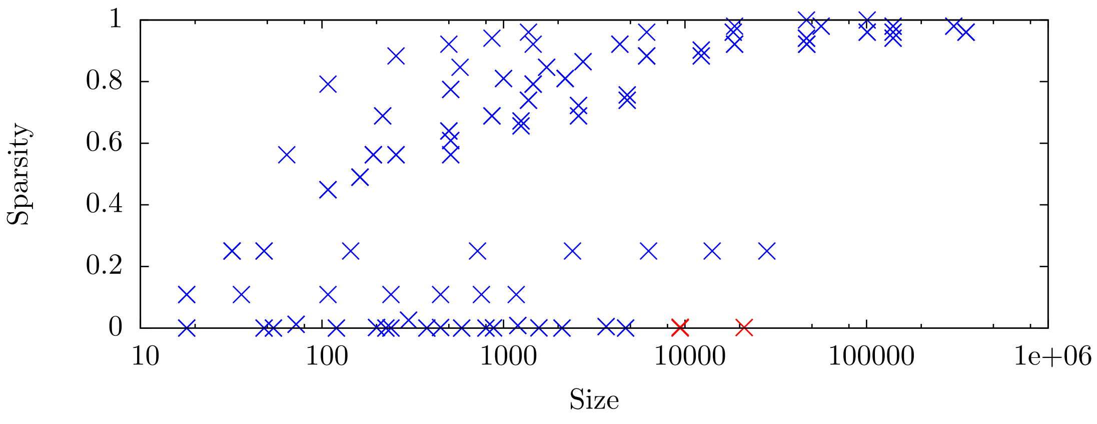


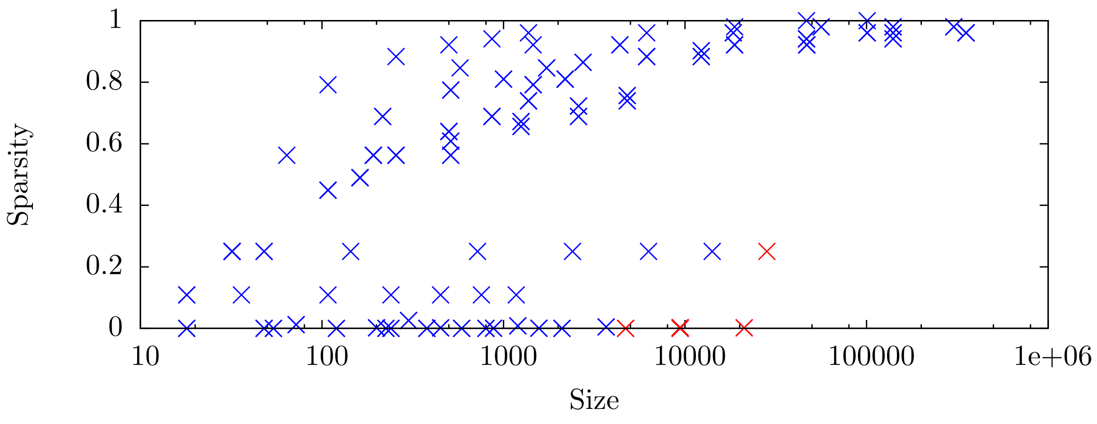
Figure F.1: Combined time for execution of a single time step in PyFR for quadrilateral elements mesh for 3rd order of accuracy and the theoretical limits imposed by the devices.

Appendix G

Performance Bounds for GiMMiK Kernels

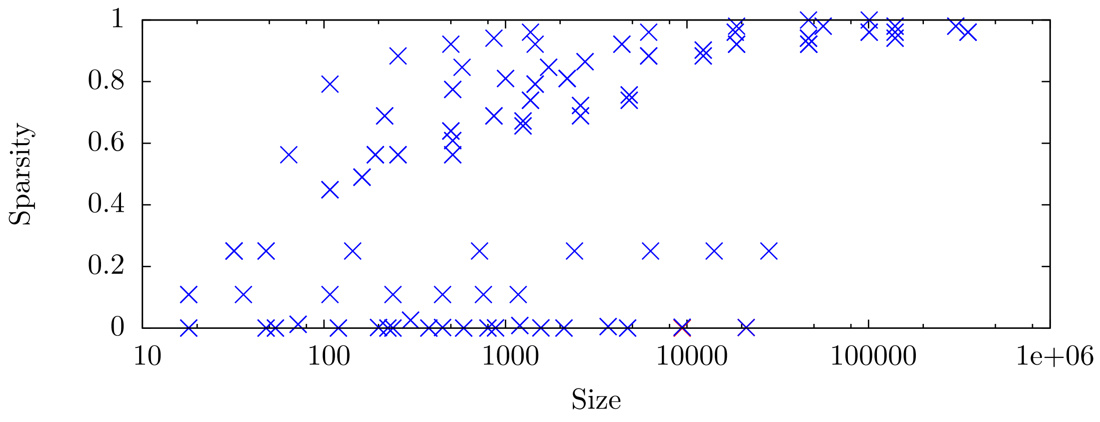


(a) Tesla K40c, $\beta = 0$

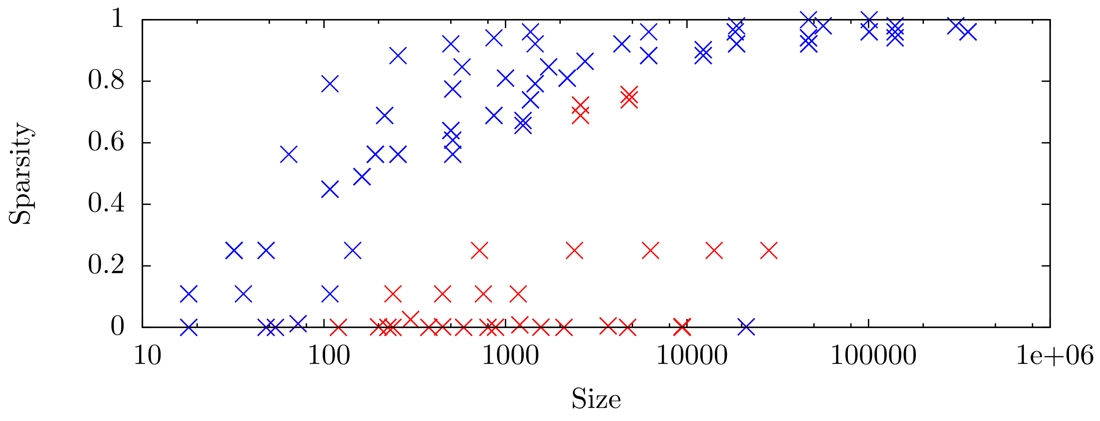


(b) GTX 780 Ti, $\beta = 0$

Figure G.1: Memory bandwidth bound (blue) and floating-point rate bound (red) single precision, $\beta = 0$ kernels for a set of benchmark matrices.

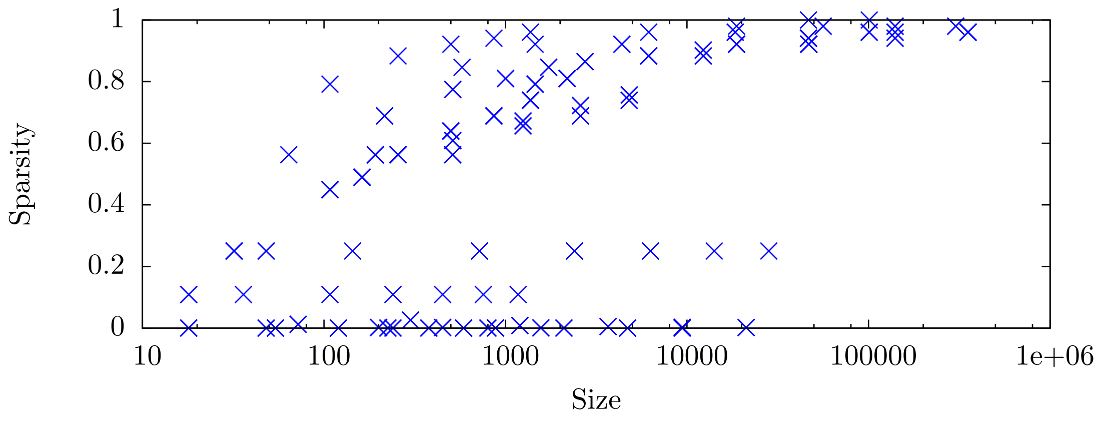


(c) Tesla K40c, $\beta \neq 0$

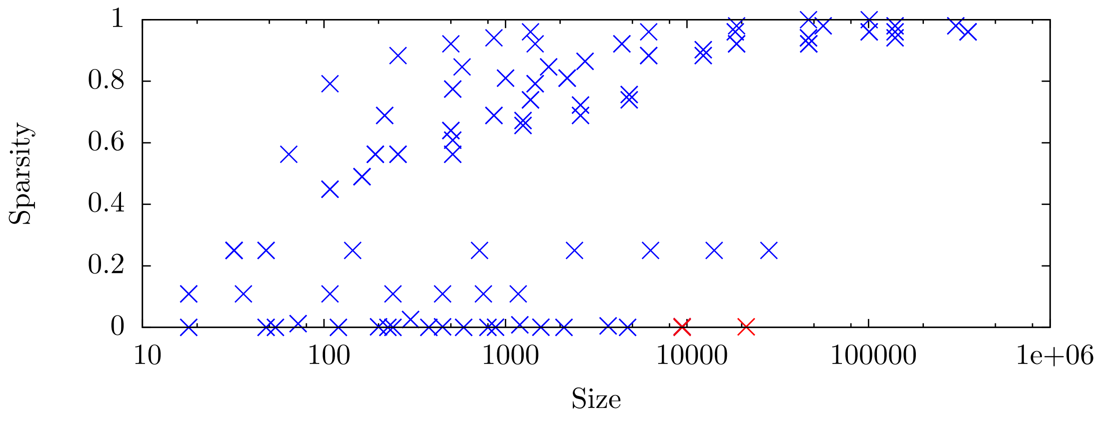


(d) GTX 780 Ti, $\beta \neq 0$

Figure G.1: Memory bandwidth bound (blue) and floating-point rate bound (red) double precision, $\beta \neq 0$ kernels for a set of benchmark matrices.



(e) Tesla K40c, $\beta \neq 0$



(f) GTX 780 Ti, $\beta \neq 0$

Figure G.1: Memory bandwidth bound (blue) and floating-point rate bound (red) single precision, $\beta \neq 0$ kernels for a set of benchmark matrices.