One Size Doesn't Fit All: Quantifying Performance Portability of Graph Applications on GPUs

Tyler Sorensen
Princeton University, USA
UC Santa Cruz, USA
tyler.sorensen@ucsc.edu

Sreepathi Pai
University of Rochester, USA
sree@cs.rochester.edu

Alastair F. Donaldson
Imperial College London, UK
afd@ic.ac.uk

***Abstract*—Hand-optimising graph algorithm code for different GPUs is particularly labour-intensive and error-prone, involving complex and ill-understood interactions between GPU chips, applications, and inputs. Although the generation of optimised variants has been automated through graph algorithm DSL compilers, these do not yet use an optimisation policy. Instead they defer to techniques like autotuning, which can produce good results, but at the expense of portability.**

In this work, we propose a methodology to automatically identify portable optimisation policies that can be tailored ("semi-specialised") as needed over a combination of chips, applications and inputs. Using a graph algorithm DSL compiler that targets the OpenCL programming model, we demonstrate optimising graph algorithms to run in a portable fashion across a wide range of GPU devices for the first time. We use this compiler and its optimisation space as the basis for a large empirical study across 17 graph applications, 3 diverse graph inputs and 6 GPUs spanning multiple vendors. We show that existing automatic approaches for building a portable optimisation policy fall short on our dataset, providing trivial or biased results. Thus, we present a new statistical analysis which can characterise optimisations and quantify performance trade-offs at various degrees of specialisation.

We use this analysis to quantify the performance trade-offs as portability is sacrificed for specialisation across three natural dimensions: chip, application, and input. Compared to not optimising programs at all, a fully portable approach provides a $1.15\times$ improvement in geometric mean performance, rising to $1.29\times$ when specialised to application and inputs (but not hardware). Furthermore, these semi-specialised optimisations provide insights into performance-critical features of specialisation. For example, optimisations specialised by chip reveal subtle, yet performance-critical, characteristics of various GPUs.

I. Introduction

Program transformations, e.g. compiler optimisations, influence the runtime performance of programs but are affected by the *environment* in which the program will run, such as the architecture, the actual program (application), and input. If these aspects are fixed, then the transformations can be *specialised*, usually by autotuning [1]–[7], often resulting in significant performance improvements.

Unfortunately, by design, such specialisation does not transfer to other environments. If the architecture, application, or even input change, then the performance impact of transformations may also change. In some undesirable cases, a transformation that caused a speedup for one environment can cause slowdowns for another. In our work, for example, a read-modify-write aggregation transformation applied to the *sg-cmb* microbenchmark (discussed in Section VIII) shows a speedup of more than $22\times$ on an AMD GPU, but yields a *slowdown* ($.88\times$) on an Nvidia GPU.

Although specialisation is useful, identifying transformations that lead to *portable* performance improvements, i.e. those that yield improvements consistently, can deliver important insights into similarities (and differences) across environments. Additionally, portable transformations can be more widely deployed, reducing the maintenance demanded by fragile specialisations. To the best of our knowledge, however, there is no systematic and automatic methodology that can identify these portable transformations from a larger set of compiler transformations. In part, this is not a straightforward problem: prior work [8]–[10] has shown that even identifying transformations that yield performance improvements in a *fixed* environment can be confounded by chance effects.

Worse, a quest in search of portable transformations may be quixotic – there may be no such set of portable transformations due to the diversity of architectures today. In that event, then, we would like a rigorous methodology that explicitly delimits the environment in which a particular transformation is effective. For example, an analysis that can confirm that transformation T only yields performance improvements on architecture A is still valuable. Knowledge of such *semi-portable* transformations would also immediately enable semi-specialisation. Specialisation does not have to be an all or nothing strategy. Transformations can be specialised across some components of the environment (e.g. the application) if they are known at the time of transformation, while remaining unspecialised and portable over others.

What we lack, however, is a methodology to precisely quantify the performance trade-off associated with specialisation vs. portability. For example, writers of domain-specific compilers might wonder whether autotuning (i.e. full specialisation) is really worth it for their domain. Could they get away with enabling a set of options "semi-specialised" to most environments they expect their compilers to generate code for? Settling a question like this would require rigorously quantifying the performance gap of such semi-specialisation over full specialisation.

In this work, we provide a methodology to resolve these questions for the common case of a (usually domain-specific)

compiler with a tunable set of transformations, where the performance of transformations is affected by applications, their inputs and the architectures they run on.

We subject our methodology to one of the most stringent experimental evaluations seen in this space. Our domain-specific compiler is a recent state-of-the-art compiler for graph algorithms on GPUs [11]. We use graph algorithms, because their performance is heavily influenced in non-trivial ways by their inputs. The same graph application running on a road network graph (large diameter, low average degree) and a social network graph (small diameter, power-law degree distribution) can experience different bottlenecks on the same hardware, confounding and throwing off simplistic search methods for portable transformations. A compiler framework ensures we can have thorough experimental coverage of all transformations as compared to evaluations that manually optimise [12]. By adapting the compiler to generate OpenCL, we are also able to run the graph algorithms on a diverse set of GPUs, vastly expanding the discourse over just Nvidia GPUs used in prior work.

Thus, for the first time, we quantify the performance portability of state-of-the-art graph applications on GPUs. Our work uses 17 graph applications (Table VII), using 3 classes of inputs (Table VIII), on 6 GPUs which span 4 vendors (Table I). These elements are described in more detail in Section VI. The transformation space considers 95 optimisation combinations involving load balancing, on-chip synchronisation, and read-modify-write (RMW) aggregation, discussed in Section V. These transformations (and our application set) exercise most features of the OpenCL standard, providing *fine-grained* evidence for the performance portability (or lack thereof) of this widely supported standard.

a) Contributions: Our contributions are:

- A large high-quality dataset on the impact of performance optimisations on graph algorithms running on a very diverse set of GPUs (Section VI). Compared to prior work, our study is the first to run high-performance graph algorithms on many non-Nvidia GPUs.
- A methodology that identifies portable optimisations from this dataset, and also allows us to delineate the scope of optimisations across architectures, applications and inputs (Section III). Our strategy avoids common pitfalls such as identifying trivial or biased optimisations (Section II-C) by using a rank-based magnitude-agnostic analysis.
- A quantification, for the first time, of the performance tradeoffs associated with specialisation vs. portability across three dimensions: applications, inputs and GPUs (Section VII).
- A methodology to identify *performance-critical* differences in GPUs using information from the portability of optimisations (Section VIII). For example, we find that an ARM GPU suffers significant performance loss from memory divergence, which can be alleviated using a semantically unnecessary barrier.

TABLE I
THE 6 GPUS CONSIDERED IN THIS STUDY, THEIR NUMBER OF COMPUTE UNITS (#CUS), THE SUBGROUP SIZE (SG SIZE) AND A SHORT NAME USED THROUGHOUT THE TEXT.

Vendor	Chip	#CUs	SG Size	Short Name
Nvidia	Quadro M4000	13	32	M4000
	GTX 1080	20	32	GTX1080
Intel	HD5500	27	8,16	HD5500
	Iris 6100	47	8,16	IRIS
AMD	Radeon R9 Fury	28	64	R9
ARM	Mali-T628	4	1	MALI

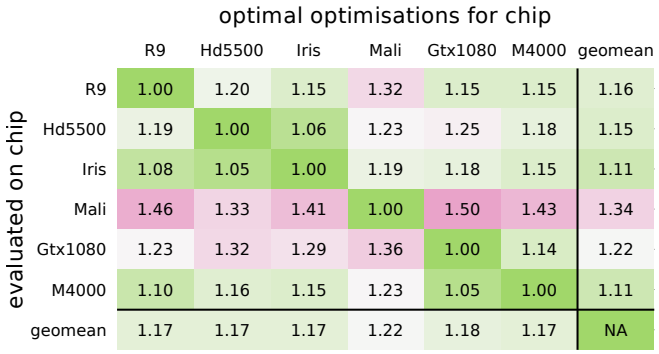

Fig. 1. Heatmap of the geomean slowdown compared to the optimal optimisations for one chip on all other chips (higher is worse).

II. INITIAL OBSERVATIONS AND MOTIVATION

While the methodology of our study is not tied to a particular domain, graph algorithms for GPUs are an interesting and pragmatic case-study given recent research interest [11], [13] and industry support [14]. In this section, we present preliminary motivation, illustrating the added complexity of considering cross-GPU portability and showing shortcomings of existing methods in the analysis of this dataset.

The GPUs of our study are summarised in Table I and described in more detail in Section VI-A: the table is shown early to facilitate discussion in this section. While we do mention optimisations, applications and inputs in this section, the discussions here do not require details apart from their names. As such, they are described more completely in Section V, Section VI-B and Section VI-C, respectively. To ease presentation: GPU names are given in SMALL CAPS; input names are given in `fixed-width`; optimisation names are given in sans-serif; and application names are given CAPS with application variants distinguished by $\text{CAPS}_{\text{subscript}}$.

A. GPUs: A Distinct Dimension

Our dataset allows us to compare the performance of optimal optimisation settings on a given GPU for an application and input when it is run with the same settings on another GPU. This allows us to study how performance varies when settings specialised to one GPU are applied to another. We

TABLE II
THE MAX SPEEDUP AND SLOWDOWN PER CHIP AND ASSOCIATED APPLICATION. THE INPUT IN EACH CASE IS `USA.NY`.

	Max Speedup		Max Slowdown	
	App	Value	App	Value
GTX1080	$SSSP_{nf}$	5.10	PR_{wl}	7.99
M4000	$SSSP_{nf}$	3.54	PR_{tp}	10.00
HD5500	$SSSP_{nf}$	16.61	PR_{tp}	22.15
IRIS	BFS_{tp}	13.25	PR_{wl}	18.70
R9	BFS_{hybrid}	14.61	$SSSP_{wl}$	6.89
MALI	BFS_{tp}	3.95	$SSSP_{wl}$	15.21

use the geomean slowdown to summarise these performance numbers across all applications, inputs and GPUs.

The results are summarised in the heatmap of Figure 1, where the rows correspond to GPUs that the programs are run on and the columns correspond to optimal optimisations for different GPUs. The diagonal is 1.00 as there are no slowdowns for a GPU using its own optimal optimisations. The values of the bottom row show the geomean across all values in the column associated with an optimisation strategy. Smaller values indicate a given optimisation strategy is more portable, causing fewer slowdowns across GPUs. The numbers of the far right column show the geomean across the row associated with the GPU. Smaller values indicate a given GPU suffers fewer slowdowns under optimisation strategies tailored for different GPUs.

Unsurprisingly, the two Intel GPUs (IRIS and HD5500) show only small slowdowns when optimal optimisations are ported between them. In contrast, Nvidia's GTX1080 (a newer architecture) slows down when settings from M4000 (an older architecture) are used. However, M4000 works well with GTX1080 settings. These counter-intuitive generational differences are concerning, for they make knowledge gained on one GPU less transferable to another. Interestingly, IRIS behaves well under the R9 strategy ($1.08\times$), but the reverse is not true ($1.15\times$). MALI suffers from the highest slowdowns, likely due to its architectural differences, i.e. it is a mobile GPU. Similarly, the optimal strategy for MALI causes the highest slowdowns on other GPUs.

We note that no optimisation setting specialised to an application, input and GPU is *completely* portable to other GPUs; any chip-specialised optimisation strategy causes at least a slowdown of $1.1\times$ (geomean) when evaluated across the other chips of our study. These results are concrete evidence that that GPUs are an *independent* portability dimension in this domain, i.e. optimisations specialised to one GPU may cause slowdowns on another.

B. Fantastic Speedups and Horrible Slowdowns

Here we examine the possible envelope of relative performance speedups and slowdowns in this domain. We first note that the maximum geomean speedup (using optimal optimisations queried from our dataset) across all GPU, applications, input combinations is $1.5\times$. While this seems low, as well as the values in the previous section appearing modest, we now show that at the extreme end of the spectrum there are significant speedups and slowdowns in this domain. Table II shows these values per GPU along with their associated applications, while the input in every case turns out to be `usa.ny`. We see that the optimisations can cause both significant speedups (up to $16\times$) and slowdowns (up to $22\times$). Thus, optimisations have significant opportunities to provide speedups despite modest aggregate values, and can dramatically punish performance if applied in the wrong environment.

If we restrict our focus to the two Nvidia chips, these results mirror prior work, which only evaluated Nvidia GPUs [11]. The top speedup is $5\times$ and the maximum slowdown is $10\times$ compared to the $16\times$ speedup and $22\times$ slowdown values seen across our cross-vendor GPUs. Thus, previous work limited to Nvidia failed to convey the full performance envelope, both in potential gains and loses, in this domain.

TABLE III
THE TOP AND BOTTOM FIVE OPTIMISATION COMBINATIONS RANKED ACCORDING TO THE NUMBER OF PROGRAMS SLOWED DOWN WHEN APPLIED OVER ALL CHIP, APP., INPUT COMBINATIONS; THE TWO MIDDLE RANKS (12 AND 26) ARE DISCUSSED IN THE TEXT.

Rank	Enabled Opts	Slowdowns	Speedups	Geomean
0	fg8	36	60	1.01
1	fg	37	58	0.98
2	wg, fg8	38	61	1.00
3	wg, fg	41	55	0.96
4	sg	41	56	1.00
12	coop-cv, oitergb	49	66	1.18
26	fg8, sg, oitergb	60	103	1.15
90	wg, coop-cv, oitergb	157	44	0.72
91	sz256, wg, oitergb	167	44	0.61
92	sz256, wg	173	23	0.60
93	sz256, wg, coop-cv, oitergb	189	35	0.54
94	sz256, wg, coop-cv	195	22	0.53

C. First Analysis Attempts

Given that optimisations specialised per chip are not completely portable, we now turn our attention to difficulties involved in identifying *portable* optimisation settings. Our dataset allows us to examine all possible optimisation combinations, of which there are 95. We query our data, applying every optimisation combination globally, i.e. across all (GPU, application, input) tuples. For each tuple, we consider the performance impact of applying the optimisation combination compared to the baseline, i.e. no optimisations. We record whether the optimisations caused a speedup, slowdown, and the magnitude of the effect. The top five, bottom five, and two middle optimisation combinations are shown in Table III, ranked by the number of tuples (out of 295) that showed slowdowns when the optimisations were applied. Using this, we now discuss several natural methods for performance portability analysis and how they fall short.

TABLE IV
THE # OF SPEEDUPS/SLOWDOWNS FOR EACH GPU WHEN APPLYING THE OPTIMISATION COMBINATIONS OF RANKS 12 AND 26 OF TABLE II

Rank	M4000	GTX1080	HD5500	IRIS	R9	MALI
12	**10/21**	**00/16**	12/03	10/02	14/02	20/05
26	22/13	13/07	17/04	10/10	21/12	20/04

a) Do No Harm: Prior work on GPU performance portability analysis used a *do no harm* approach, where optimisations were only selected if they did not cause a slowdown across any of the programs [7]. Table III shows that this approach does not work in our domain. The optimisation combination causing the fewest slowdowns (rank 0), still causes slowdowns in 36 tuples. Thus, the do no harm approach would suggest using the baseline, i.e. no optimisations. While this is the safest approach across our dataset, it is underwhelming in that the high speedups of Table II are immediately unreachable.

b) Fewest Slowdowns: The natural extension to the do no harm approach, is *harm the fewest*: choose the optimisation strategy that causes slowdowns in the fewest tuples. This would also correspond to the rank 0 optimisation. This optimisation strategy also has drawbacks: the geomean speedup across all tuples is a mere $1.01\times$, and the highest individual speedup (not shown in table) is an underwhelming $2.03\times$.

c) Maximise Geomean: We then may seek the optimisation combination with the highest geomean globally; this would be the rank 12 optimisation with a geomean of $1.18\times$. This approach has a subtle flaw. Table II shows that certain GPUs are more sensitive to optimisations and yield larger speedups than others. While the geomean is more robust to outliers than the arithmetic mean, it is still a magnitude-based metric. Indeed, if we look at the per-chip speedups and slowdowns from this optimisation, shown in Table IV, we see that this strategy is easily *biased*, yielding no speedups on GTX1080 and over twice as many slowdowns as speedups on M4000. Any magnitude-based analysis will be biased towards sensitive GPUs (or applications or inputs).

d) Our Approach: Our method, described in Section III, uses a magnitude-agnostic rank-based analysis and identifies the rank 26 optimisation combination in Table III as the most portable. While it does not produce the fewest slowdowns, or highest geomean, Table IV shows that it avoids the bias against GTX1080 and M4000 and has a respectable individual maximum speedup of $6.41\times$.

III. PORTABLE OPTIMISATION STRATEGIES

The details of the optimisations considered in this work will be discussed in Section V; however our main contribution deals with reasoning about general effects of optimisations when deployed in diverse environments. Thus, we speak about optimisations abstractly in this section.

At a high-level, an *optimisation strategy* maps a tuple of the form (application, input, chip)[1] to a set of optimisations (i.e. compiler flags). To concretise what we mean by portable optimisation strategy, we formalise these strategies as functions (Table V). These functions cover all 8 combinations of specialisations over application, input, and chip, plus an additional *baseline* function. Each function defines an eponymous *optimisation strategy* that maps an (application, input, chip) tuple to an *optimisation configuration*: a set of enabled optimisations. The optimisations considered in this study are binary (on/off), with the exception of the fg optimisation and the workgroup size, which take numeric values (see Section V). In this work, we consider two choices for fg, recorded as mutually exclusive binary optimisations fg1 or fg8. Similarly, workgroup sizes of 128 or 256 are considered.

[1]We use chip instead of GPU to include the runtime environment.

TABLE V
FUNCTIONS THAT DEFINE OPTIMISATION STRATEGIES. PARAMETERS: a – APPLICATION, i – INPUT AND c – CHIP. DON'T-CARE ARGUMENTS, IGNORED BY THE FUNCTION, ARE DENOTED BY *.

Specialised Function	Output Optimisation
`baseline(`a`=⋆, `i`=⋆, `c`=⋆)`	none
`global(`a`=⋆, `i`=⋆, `c`=⋆)`	best on average
`app(`a`=A, `i`=⋆, `c`=⋆)`	best for app. A
`input(`a`=⋆, `i`=I, `c`=⋆)`	best for input I
`chip(`a`=⋆, `i`=⋆, `c`=C)`	best for chip C
`chip_app(`a`=A, `i`=⋆, `c`=C)`	best for chip I and app. A
`input_app(`a`=A, `i`=I, `c`=⋆)`	best for input I and app. A
`chip_input(`a`=⋆, `i`=I, `c`=C)`	best for chip C and input I
`oracle(`a`=A, `i`=I, `c`=C)`	best for C, I, and A

Some of these functions are easily defined. For example, the *baseline* optimisation strategy disables all optimisations, so it maps all (application, input, chip) arguments to the empty optimisation configuration. In effect, *baseline* is not specialised. At the other extreme, the *oracle* function maps each (application, input, chip) tuple to the optimisation configuration that delivered the highest speedup. It is constructed by exhaustively examining performance data from our experiments.

In this section, we present a method to construct partially specialised functions. Consider the function *app*, which specialises for application and ignores the input and chip parameters. It can be described as selecting those optimisations that are "best" for an application across all inputs and all chips. Such specialisations can also be extended into multiple dimensions. The function *chip_app*, which specialises for chip and application, returns those optimisations that are "best" for a particular application on a particular chip across all inputs.

This section describes the methodology to systematically construct these functions using performance data. Our results then examine: (1) how each of these strategies fares against the *oracle* strategy, i.e. how much is lost without specialisation in particular dimensions (Section VII); and (2) performance critical differences between GPUs that are exposed when comparing per-chip specialised optimisation configurations (Section VIII).

A. Optimisation Strategies by Specialisation

We now describe our analysis that produces, from empirical data, a spectrum of optimisation strategies between baseline

and oracle, by incorporating more and more test information. A key challenge here is to soundly identify useful optimisations – those that impact positively on performance on average – under various degrees of specialisation.

At a high level, to determine whether an optimisation should be enabled or not for a (set of) test(s), the empirical data is split into two sets. The first contains timings when the optimisation was enabled and the second when it was disabled. We then use a statistical procedure, the Mann-Whitney U (MWU) test [15], to determine whether the optimisation caused a change in test runtimes. If so, then we compare the runtime medians to determine the direction of change. Only in the case of a statistically significant speedup do we enable the optimisation.

To specialise the optimisation configurations per chip, application, input or combination of these, we perform the above analysis, but on partitions of our data. For example, when specialising optimisations per chip, we partition the data into subsets, one for each chip, and run the aforementioned procedure on each partition. Each partition will yield an optimisation strategy specialised to that partition's chip.

The analysis is shown in detail in Algorithm 1, with specialisation illustrated for the per-chip case (other specialisations are similar). We describe our approach top-down starting with function SPECIALISE_FOR_CHIP (line 1). This function simply iterates through the available chips in the global list CHIPS (line 3), partitions the data into a set per chip (line 4), and then launches the analysis for the partition, which returns an optimisation strategy recommended for the partition (line 5).

The function that identifies an optimisation configuration for a partition is OPTS_FOR_PARTITION (line 7). Here, we aim to extract the effect of an individual optimisation opt across the entirety of the data partition. To do this, we construct two lists A and B per optimisation (line 10). The lists are populated by iterating through all the valid optimisation settings where opt is enabled, returned by ALL_OPT_SETTING (line 11).

For each optimisation setting, we create the mirror setting where opt is disabled (line 12), thus the only difference between os and dis_os is that opt is enabled in the former and disabled in the latter. We then iterate through all test data in $partition$ (label 13), checking for each test p if the difference in runtime under the two optimisation configurations is significant (using the 95% CI) via SIGNIFICANT (line 14). If so, then we normalise the runtime with optimisation enabled to the runtime when it was disabled and add it to the list A (line 15), while adding 1.0, the normalised baseline, to the list B (line 16).

With data from A and B, we can now perform a statistical analysis to determine if we should enable opt (lines 17 and 18). Essentially, ENABLE_OPT (line 20), takes the two lists A and B and applies the MWU test (line 21). The MWU test assesses whether there is a statistical difference in *ranks* between the two lists, i.e. if one list has stochastically larger values than the other. If the test confirms with $p < .05$, we check the median of A to determine whether we enable the optimisation.

While we could use a variety of statistical methods on the normalised runtimes, the MWU test has the property of being rank-based, i.e. it does not consider the magnitudes of value differences. As we saw in Section II-C, magnitude-based approaches can produce biased results.

Algorithm 1 Finding optimisation strategies.

```
 1: function SPECIALISE_FOR_CHIP(data)
 2:     chip_opts = MAP()
 3:     for chip in CHIPS do
 4:         chip_data = {X in data where X was run on chip}
 5:         chip_opts[chip] = OPTS_FOR_PARTITION(chip_data)
 6:     return chip_opts

 7: function OPTS_FOR_PARTITION(partition)
 8:     enabled_opts = {}
 9:     for opt in OPTS do
10:         A = [], B = []
11:         for os in ALL_OPT_SETTINGS(opt) do
12:             dis_os = os[opt = disabled]
13:             for p in partition do
14:                 if SIGNIFICANT(p[os], p[dis_opt]) then
15:                     A.add(p[os] / p[dis_opt])
16:                     B.add(1.0)
17:         if ENABLE_OPT(A,B) then
18:             enabled_opts.add(opt)
19:     return enabled_opts

20: function ENABLE_OPT(A,B)
21:     reject_null = MWU(A,B)
22:     return reject_null and MEDIAN(A) < 1.0
```

IV. OpenCL Background

In this section we provide necessary background for the discussions in Sections V to VII, namely we describe the OpenCL programming model and how the programming constructs map to generic GPU architecture features.

An OpenCL program has two parts: code that is executed on the *host* (e.g. the CPU system), and code that is executed on the *device* (e.g. a GPU). The host code is written in C/C++, and is responsible for setting up the device memory and calling device programs (called *kernels*), via an API. The device code is written in OpenCL C [16], which is derived from C99. The code is executed in an SPMD (single program, multiple data) manner. That is, each thread executes the same program, but has access to unique identifiers that can be used to guide different threads to different data or program locations.

A. OpenCL Hierarchy

OpenCL C supports a hierarchical programming abstraction with components that map naturally to GPU architectural features. At the base level there are *threads*, which are partitioned into *subgroups*. Threads in the same subgroup are often executed on the same hardware vector unit. Multiple subgroups make up a *workgroup*. Typically subgroups in the same workgroup are executed on the same *compute unit* or CU, a hardware resource that can execute several subgroups concurrently. A kernel is executed by one or more workgroups.

TABLE VI
OPTIMISATIONS AND THEIR PERFORMANCE PARAMETERS

Optimisation	Performance Parameters
coop-cv	workgroup size, subgroup size, atomic read–modify–write throughput, subgroup collectives throughput
fg	local memory, workgroup-barriers
sg	subgroup size, subgroup-barrier throughput, local memory constraints
wg	workgroup size, local memory constraints, workgroup-barrier throughput, workgroup atomic load/store throughput
oitergb	kernel launch and host-device memory transfer overhead, global synchronisation, inter-workgroup scheduler
sz256	occupancy, workgroup-local resource limits

a) OpenCL Memory Regions: The memory region shared at the highest level of the hierarchy is *global memory*. All threads executing a kernel can access the device's global memory; threads can perform ordinary reads from and writes to this memory, along with a variety of read-modify-write instructions (e.g. compare-and-swap). Threads in the same workgroup can communicate (using the same operations) through faster *local memory*, typically provided in a CU-local cache.

Finally, threads have a private memory region where thread-local values can be stored, typically provided in a register file. Many GPUs now also provide subgroup communication primitives, so threads in the same subgroup can efficiently communicate values in private memory.

b) Synchronisation: GPU synchronisation mechanisms include the intra-workgroup barrier, in which threads in the same workgroup can synchronise via a primitive instruction and the more recent, subgroup barrier, which allows threads in the same subgroup to synchronise. These barrier instructions are required to be convergent, i.e. executed by all the synchronising threads, or none of them. Thus, if these instructions appear in a loop, it should be ensured that the loop is uniform across workgroups or subgroups, depending on the barrier flavour. OpenCL 2.0 provides a detailed *memory consistency model*, similar to C++, which allows fine-grained communication in a well-defined, race-free manner through special atomic instructions, which provide building blocks for useful synchronisation idioms (e.g. mutexes).

The OpenCL standard does not provide any independent forward progress guarantees between threads in different workgroups. This means that in principle, blocking synchronisation, such as inter-workgroup barriers, are prone to unfair executions that lead to threads being blocked indefinitely. However, prior work has shown that current GPUs do provide a degree of forward progress under the *occupancy bound execution model* which states that concurrently executing threads will continue to be concurrently executed [17]–[19]. Several applications and optimisations in our study rely on this assumption.

V. GENERALISING OPTIMISATIONS TO OPENCL

Recent work on GPU acceleration of graph algorithms presented four key architecture-independent graph algorithm optimisations, which were shown—via their embedding in an optimising compiler generating CUDA code—to achieve state-of-the-art performance for a number of applications running on Nvidia GPUs [11]. To study performance portability, we generalise the key optimisations to OpenCL and retarget this compiler, which generates only CUDA, to generate OpenCL.

We discuss the four optimisations in Sections V-A to V-D, in each case providing: a description of the optimisation; challenges associated with generalising the optimisation to OpenCL; and features that govern the performance potential. The optimisations are summarised in Table VI.

A. Cooperative Conversion

The OpenCL execution hierarchy can be exploited to reduce the cost of expensive, serialising operations such as atomic read-modify-write (RMW) instructions. For example, many graph algorithms track the dynamic workload through a global worklist. Each push to this worklist by a thread ordinarily requires one RMW. However, threads can communicate at the subgroup (or workgroup) levels to combine individual pushes into a single push of multiple items that uses only one atomic RMW. We abbreviate this optimisation to coop-cv.

a) OpenCL Generalisation: Unlike in CUDA, OpenCL subgroups do not have to execute in lockstep.[2] Therefore, subgroup operations must be *uniform*, i.e. they are executed by all the threads in the subgroup or none of them. Thus, our compiler must generate uniform branches (e.g. by equalising loop trip counts across threads) and use predication to prevent execution of code that would ordinarily have not executed.

b) Performance Considerations: The number of elided atomic RMW operations depends on workgroup size and subgroup size. Communication uses local memory and the appropriate barriers (workgroup/subgroup). Note that on architectures that implement subgroups with lockstep execution, subgroup barriers are free. The performance impact of this optimisation depends on the the overhead of the orchestration vs. the cost of the global RMW operations.

B. Nested Parallelism

Not to be confused with OpenCL Nested Parallelism [20, Sec. 3.2.3], which mimics CUDA's Dynamic Parallelism [21, App. D], the nested parallelism optimisation tackles the classic problem of parallelising nested loops, the inner of which is usually irregular in graph algorithms. Specifically, it generates *inspectors* and *executors* that inspect the inner loop iteration space at runtime and redistribute work among the threads. The specific schemes for distributing work are based on proposals by [22] and redistribute work among threads of the workgroup (wg) or threads of the subgroup (sg). When redistributing to threads of the workgroup, the executor can choose between serialising the *outer* loop or linearising the iteration space

[2]Starting with Nvidia Volta, CUDA warps do not execute in lockstep either.

(fine-grained or fg). Often, all three strategies wg, sg or fg must be used in combination, with wg handling high-degree nodes, sg handling medium-degree nodes and fg handling the rest.

The fg variant can also be parameterised by the number of edges processed per iteration. We consider two possibilities in this work: a single edge (denoted fg1) and eight edges (denoted fg8). The entire class of nested parallelism optimisations is abbreviated to np.

a) OpenCL Generalisation: The fg scheme of this optimisation is straightforwardly ported to OpenCL. The wg scheme scheme requires concurrent writes to the same location in a leader-election idiom. OpenCL deems this as data-race, rending the entire program undefined. Thus, we identified all race-y accesses and change them to OpenCL 2.0 atomic operations. The sg scheme requires OpenCL subgroup adaptations, similar to coop-cv.

b) Performance Considerations: Because these schemes involve inter-thread communication mediated through barriers (both workgroup and subgroup), the throughput of barriers is a critical factor. Communication and work redistribution occur through local memory so its read and write latency is important. Finally, if there is very little load imbalance among threads (for example, due to uniform degree graphs), these schemes simply add overhead.

C. Iteration Outlining

Many graph algorithms execute kernels iteratively until a fixed point has been reached. In breadth-first search (BFS), the number of dependent iterations is proportional to the diameter of the graph, which in the case of planar graphs like road networks, may be thousands of iterations. If each kernel execution is very short, then the launch overhead dominates execution time. In iteration outlining, code that launches kernels is *outlined* to the GPU. As a result, kernel launches are turned into GPU function calls, with synchronisation between function calls provided by a global barrier. We abbreviate this optimisation to oitergb (*o*utline *iter*ations using a *g*lobal *b*arrier).

a) OpenCL Generalisation: The crux of this OpenCL implementation is a *portable* global barrier. Although global barriers have been proposed on a per-GPU basis, e.g. for CUDA [19], they cannot be used as is, since OpenCL barriers need to be (functionally) portable. Current GPUs do not provide forward progress properties across all threads and a non-portable global barrier implementation can hang. To provide a portable global barrier, we follow the recipe given in [17], which involves dynamically discovering the occupancy of the GPU at runtime and creating a custom execution environment in which a barrier can be executed.

b) Performance Considerations: The performance impact of iteration outlining on a platform depends on the relation between the kernel launch overhead (including a memory copy), the execution time of the barrier and the execution time of the kernel. While the first two are largely architecture-dependent, the last also depends on the application and input.

TABLE VII
THE 17 APPLICATIONS CONSIDERED IN THIS STUDY ALONG WITH THEIR APPLICABLE OPTIMISATIONS. APPLICATION VARIANTS CONSIDERED STATE-OF-THE-ART ARE NOTED WITH A (*)

App	Variant	Available Opts
BFS	cx, topo, tp, wl, hybrid*	sz256, np, coop-cv, oitergb
CC	*	sz256, np, coop-cv, oitergb
MIS	worklist, pannotia*	sz256, coop-cv, oitergb
MST	single-wl, multiple-wl*	sz256, coop-cv
PR	residual-wl*	sz256, np, coop-cv
	residual, tp	sz256, np
SSSP	topo	sz256, coop-cv, oitergb
	wl, nf*	sz256, np, coop-cv, oitergb
TRI	*	sz256, np

TABLE VIII
THE 3 GRAPH INPUTS CONSIDERED IN THIS STUDY, ALONG THE NUMBER OF NODES, EDGES, AVERAGE DEGREES AND DIAMETER

Input	Nodes	Edges	Avg/Max Degree	Diameter
usa.ny	264346	730100	2.76/8	620
rmat20	1048576	8259994	7.88/1181	13
2e23	8388608	33554432	4.00/16	18

D. Workgroup Size

The final optimisation we consider is simply resizing the number of threads in the workgroup from 128 to 256. We choose 128 as the default as not all of the GPUs we investigate support the workgroup size of 256 (see Section VI-A). The workgroup size is known to affect occupancy, which can affect performance. Functionally, this optimisation requires that all kernels be agnostic to the workgroup size. Our compiler only generates kernel with this property, thus implementation is straightforward. We abbreviate this optimisation to sz256; when the optimisation is disabled, the workgroup size is 128, and when it is enabled, the workgroup size is 256.

E. The Optimisation Space

With the exception of fg and sz256, all optimisations can be enabled or disabled independently. In the case of fg, we consider two variants, fg1 and fg8. In the case of sz256, we consider two values: 128 or 256. Thus, there are 95 total optimisation combinations, excluding the baseline, where no optimisations are enabled.

VI. GPUs, Apps and Inputs of the Study

We now detail the scope of our study: the different GPUs, applications and inputs for which performance data were gathered. Formally, an *application* is a graph algorithm expressed in our DSL. An application accepts an *input*, which is a graph. A *chip* refers to a GPU, but also includes the runtime environment. The GPUs, applications and inputs used in this study are summarised in Tables I, VII and VIII, respectively. We now provide an overview of each dimension.

A. GPUs Studied

Table I summarises the GPU platforms we use for our evaluation: 6 GPUs spanning 4 vendors. The AMD R9 GPU and the Nvidia GPUs are discrete; all others are integrated. GPUs from the same vendor also span different architecture configurations: the Nvidia M4000 and GTX1080 GPUs belong to the Maxwell and Pascal architectures respectively; the Intel HD5500 and IRIS GPUs both use the Broadwell architecture, but at different graphics tiers (GT2 and GT3, respectively). Experiments for the Intel and AMD GPUs were run on the Windows OS, whereas Linux was used for the Nvidia and ARM GPUs.

Due to varying support of OpenCL versions, the GPUs of Table I do not all support the features required for our optimisations (Section V). However, using various GPU-specific functionality, we were able to mimic the required memory model and subgroup support.

The ARM and Nvidia chips do not natively support the OpenCL 2.0 memory consistency model. Thus, we needed to add atomic load/store operations and memory fences, as required by the specification. For Nvidia, we use inline PTX memory fences [23] to provide OpenCL 2.0 atomics as in prior work [24]. On ARM, because there is no low-level inline language support, we provide a best-effort implementation using memory fences provided in early versions of OpenCL (1.x) and validate our results against an oracle implementation.

Only the AMD GPUs support OpenCL subgroups natively. For Intel GPUs, we use their specialised subgroup extension [25] that provides the needed functionality. On Nvidia, whose hardware supports subgroups, though its OpenCL implementation does not, we use inline PTX to access the equivalent warp intrinsics [23, p. 209]. Our ARM GPU does not support subgroups [26], so we default to a subgroup size of one, which is a trivial, but semantically valid.

No GPU guarantees the forward progress required by the blocking synchronisation used in our programs (i.e. global barriers, and mutexes). However, prior work has demonstrated that the GPUs in our study empirically support a limited form of forward progress [17] sufficient to implement our required synchronisation idioms.

B. Applications

The IrGL compiler is accompanied by 19 graph applications, of which we are able to use 17. We do not use Delaunay Mesh Refinement (DMR) or SSSP (priority worklist variant) since some of their (large) support libraries are written in CUDA with no simple OpenCL alternative. The applications can be split into 7 high-level problems – Breadth-first search (BFS), Connected Components (CC), Maximal independent set (MIS), Minimal spanning tree (MST), PageRank (PR), Single-source shortest path (SSSP), and Triangle counting (TRI). Each problem has multiple implementation strategies, summarised in Table VII, the strategies marked (*) implement the fastest algorithms. An empty "Variants" column indicates that there is only one variant of the algorithm. The "Available Opts" column shows which optimisations apply to an implementation – we generate variants with all possible combinations of these optimisations for our study. Each application is accompanied by a checker to validate executions.

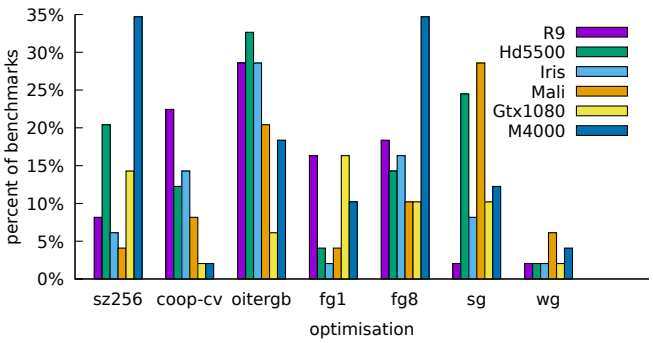

Fig. 2. Summary of optimisations necessary for top speedups per chip.

C. Inputs

We use inputs from three classes: a road network of New York (`usa.ny`), a uniformly random graph (`2e23`), and a RMAT-style graph (`rmat20`) [27]. The `usa.ny` road network is a high-diameter, uniform low degree graph. The `2e23` is synthetic random graph with uniformly distributed edges. `rmat20` is also a synthetic graph, but constructed using a recursive procedure, so that vertex degrees exhibit a power-law distribution [27]. We did not run MST on `2e23` due to an undiagnosed error.

D. Gathering Data

For our results, we record the time it takes to execute the application on the GPU. Because the optimisations we consider are only related to graph computation, and not file IO or memory transfers, we ignore time taken to load the graph inputs or the initial and final transfers of graph data to and from the GPU. Indeed, this is the approach taken in the original presentation of these optimisations [11].

We run each test three times. We compute the average and 95% confidence intervals [28]. In total, out of the 295 tests, we were unable to observe speedups using any optimisation combination in 127 (43%). These tests are fairly evenly distributed across chips. Thus some application, input combinations are not sensitive to our optimisations.

In total, across all platforms and benchmarks, our experimental run takes 237 hours. The slowest platform is MALI, taking 97 hours as the GPU is significantly smaller, having a clock rate of only 533 MHz and 4 compute units.

E. Optimisations Across GPUs

Given that the optimisations of Section V were originally developed for, and have only been tested on Nvidia GPUs [11], we now assess these optimisations for cross-vendor utility. To do this, we find the optimisation combination (call it O) which yields the lowest runtime for each chip, application, input combination (call the combination T). We then examine

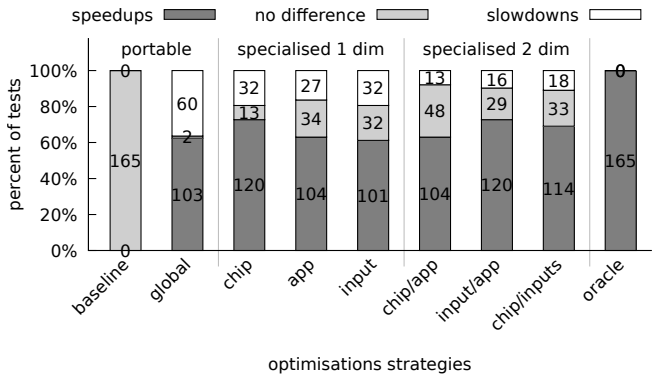

Fig. 3. For each degree of specialisation, the percentage of tests that the corresponding optimisation strategy provided: a speedup, no difference or slowdown. Test counts are given on the bars.

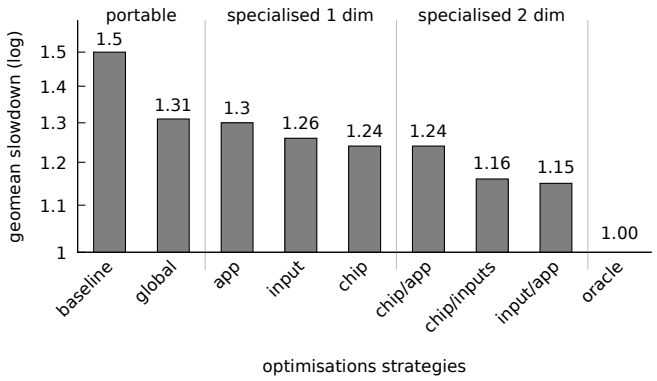

Fig. 4. The geomean slowdown compared to the oracle across all tests for different opt. strategies. Concrete test counts are given on the bars.

each individual optimisation o enabled in O; if removing o from O causes a statistically significant slowdown (discussed in Section VI-D) in T, then we say o is *necessary* for the top speedup in T. Figure 2 shows, for each optimisation, the percentage of top speedups that the optimisation is necessary for. To show cross-vendor utility, we show optimisations split per chip. Although the percentages vary, every optimisation is necessary for some top speedups across all chips. Thus, these optimisations are not specific to Nvidia GPUs.

VII. Quantifying Semi-Specialisation

We now present the results of applying our analysis of Section III-A to our data for various degrees of portability. This allows us to quantify the trade-off whereby performance improves as portability is reduced. We show summary results relating to portability vs. specialisation for our test set, considering three dimensions of specialisation: chip, application, and input. To specialise over a dimension d, we partition our tests into distinct subsets where all tests in a subset have the same value for d. Each subset is then assigned an optimisation strategy using our analysis of Section III-A.

The concrete per-chip strategy is discussed in Section VIII; other strategies are elaborated on in [29, Ch. 4].

The Effects of Specialisation: We start our results discussion with Figure 3 which shows, for each optimisation strategy, the percentage of tests that exhibited a speedup, slowdown or no significant change under the optimisation strategy. In this chart we exclude the 43% of tests for which we did not observe speedups (see Section VI-D). As a result, the baseline strategy shows no difference on all tests and the oracle strategy shows speedups on all tests.

The completely portable strategy (global) provides a speedup on 62% of the tests and a slowdown on 18% of tests (compared to baseline). Each additional dimension of specialisation roughly halves the number of slowdowns.

The number of speedups does not appear to follow the same trend, even occasionally decreasing as portability is reduced, e.g. when application or input specialisation is added to the chip specialisation, the number of speedups decreases. However, in these cases, the number of slowdowns shows a larger decrease; this decrease in slowdowns is where the benefit of specialisation occurs.

While Figure 3 shows the number of speedups and slowdowns, it does not measure the magnitude of the runtime differences between optimisation strategies. Figure 4 shows the geomean slowdown over all tests normalised to the oracle.

Using both Figure 3 and 4, we make the following observations: the optimal single dimension to specialise for speedups is chip, which provides 120 speedups as opposed to 104 and 101 for app and input respectively. Additionally, the geomean slowdown is $1.24\times$ as opposed to $1.3\times$ and $1.26\times$ for app and input. Specialising for applications gives the fewest slowdowns but also the largest mean slowdown.

On the other hand, the optimal two dimensions to specialise across for speedups is inputs and applications, with 120 tests showing a speedup with a geomean slowdown of $1.15\times$. While the single dimension chip specialisation gives the same number of speedups, it has twice as many slowdowns (16 vs. 32). The optimal two dimensions for the fewest slowdowns is applications and chip, with 13 slowdowns. This optimisation strategy also has the largest geomean slowdown at $1.24\times$. Interestingly, this is the same geomean slowdown as the optimisation strategy for only chips. This suggests that the chip and application dimensions do not synergise well.

VIII. Dissecting Chip-specific Optimisations

The `chip` function selects optimisations solely based on chip and can be used to identify performance-critical differences between GPUs. Table IX contains the full `chip` function obtained from our analysis, along with the effect size reported by the MWU test. Recommendations by our analysis to enable an optimisation for a chip are marked with a ✓. Similarly, recommendations that an optimisation should not be enabled (because it is ineffective or hurts performance on that chip) are marked with ✗ using a lighter font color. In one case (fg8 on Mali), there are not enough results with statistically significant differences for the analysis to make a confident decision (i.e. with $p < .05$).

TABLE IX
THE OPTIMISATION RECOMMENDATIONS BY OUR ANALYSIS PER CHIP, DEFINING THE CHIP FUNCTION OF TABLE V. ENTRIES ARE GIVEN WITH THEIR COMMON LANGUAGE EFFECT SIZE AND MARKED WITH A ✓(✗) TO INDICATE THAT THEY ARE RECOMMENDED TO BE ENABLED (DISABLED). ONE OPTIMISATION IS UNCERTAIN (MARKED WITH ?). WE OMIT SZ256 AS IT IS NOT RECOMMENDED FOR ANY CHIP. WE OMIT FG1 AS IT IS RECOMMENDED IN EVERY CASE THAT FG8 IS RECOMMENDED AND IS LESS EFFECTIVE (ACCORDING TO MEDIAN SPEEDUP).

Chip	coop-cv	oitergb	fg8	sg	wg
R9	✓.70	✓.65	✓.90	✓.74	✗.18
HD5500	✗.41	✓.65	✓.54	✓.56	✗.11
IRIS	✓.67	✓.73	✓.58	✓.63	✗.09
MALI	✗.12	✓.71	?.47	✓.76	✗.12
GTX1080	✗.19	✗.22	✓.86	✓.78	✗.32
M4000	✗.07	✗.47	✓.86	✓.68	✗.22

Table IX also lists the *common language effect size* [30] (CL) for an optimisation on each chip. The CL denotes the probability a randomly chosen program and input pair will show a speedup for an optimisation on a particular chip. Recall that our definitions of speedup and slowdown require statistically significant differences with 95% confidence.

From the effect size, we see that fg8 nearly always provides a speedup on AMD and Nvidia GPUs (with over an 85% probability). On Intel GPUs, fg8 also provides speedups, but it is not as widely applicable – less than a 60% probability. While all chips enable sg, the effect size varies from between .56 and .78. The effect size for wg is low for all chips, however it is non-zero, meaning there are some cases where it provides a speedup (as seen in Figure 2). Interestingly, fg8 on MALI and oitergb on M4000 have the same effect size (.47), but the test was not confident on the former. This is because there were not as many samples with significant differences in the comparison sets for the MALI. Finally, although we ran each test only three times, this was sufficient for our analysis to make confident recommendations on all but one of the chip, optimisation query pair.

We now demonstrate how differences in the recommended optimisations can reveal insights about different chips. In particular, we focus on why the strategies for: (1) the two Nvidia chips do not enable oitergb (2) IRIS and R9 enable coop-cv, and (3) MALI enables sg, even though it does not have (physical) subgroups.

a) Overhead of Kernel Launches and Memory Copies: The oitergb optimisation is enabled by all chips except those from Nvidia. We investigate the cause by using a microbenchmark similar to the one used to motivate the optimisation on Nvidia architectures [11]. Essentially, this launches a constant-time kernel a fixed number of times (10000), interleaving the launches with a memory copy of a single integer from the GPU to the CPU. The constant-time kernels establish the exact utilisation of the GPU, so timing the entire procedure reveals the this overhead of launching these kernels and of the memory copies that oitergb is designed to reduce. OpenCL does not provide device timers, thus we use a calibration loop

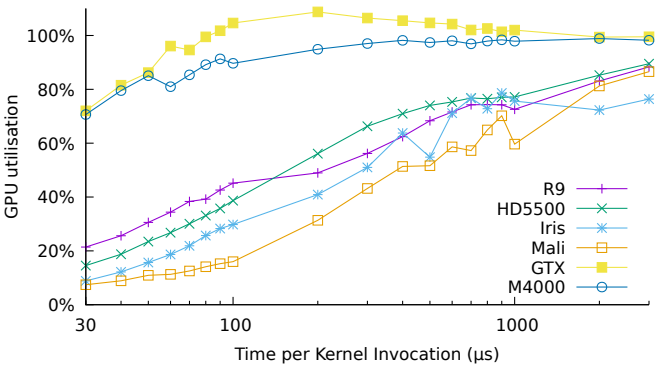

Fig. 5. Per-chip results of the kernel launch frequency microbenchmark.

TABLE X
MICROBENCHMARK RESULTS FOR SUBGROUP ATOMIC COMBINING (*sg-cmb*) AND WG MEMORY DIVERGENCE (*m-divg*).

	R9	HD5500	IRIS	MALI	GTX1080	M4000
sg-cmb	**22.10**	.98	**7.95**	1.06	.88	.97
m-divg	1.04	1.07	1.08	**6.45**	1.27	1.08

and therefore our results are somewhat noisy.

Figure 5 shows the utilisation of the GPU as the kernel execution time is varied. For a given kernel time, we can see that Nvidia chips have relatively higher utilisation than other chips, implying that they have the lowest launch and memory copy latencies. The kernel launch and memory copy overheads are sufficiently higher for all other chips that they need oitergb for performance. Note that oitergb is used on Nvidia GPUs (Figure 2), but for fewer benchmarks than other chips.

b) Subgroup Atomic RMW Combining: The coop-cv optimisation is enabled for R9 and IRIS (but not the other Intel chip, HD5500). The most common form of this optimisation aggregates atomic RMW instructions within a subgroup. To investigate why this optimisation is only turned on for a few architectures, we wrote an OpenCL microbenchmark to measure the time for N `atomic_fetch_and_add` invocations on a single memory location (here, $N = 20000$). We wrote a separate microbenchmark that measures the time after combining all atomics in the subgroup into one atomic (mimicking coop-cv), thus potentially improving throughput by the size of the subgroup. The *sg-cmb* row of Table X shows the speedup of this coop-cv version over the original.

The speedups from R9 and IRIS, the two chips for which our analysis suggest the coop-cv optimisation, are notably higher than values for the other chips. The overhead of subgroup communication for combining causes the speedups to be a fraction of the subgroup size – R9 has a subgroup size of 64, but sees only a 22× speedup. IRIS uses a subgroup size of 16 for these kernels, and delivers about half of that as speedup.

MALI has a subgroup size of 1, and does not show speedup as expected. The Nvidia chips and HD5500 do not exhibit speedups, but on further investigation we find that their respective OpenCL JIT compilers already implement the coop-

cv optimisation for subgroups.

c) Intra-workgroup Memory Divergence: Since MALI has a trivial subgroup size of 1, we were intrigued to find that sg is enabled for it. Recall that sg improves load balance by redistributing work over threads from the same subgroup. By careful elimination, we discovered that workgroup barriers placed to separate sg execution from the rest of the kernel were the source of the speedup. Previous work [31] has found that gratuitous barriers can reduce the *memory divergence* and thus, improve performance.

We built a microbenchmark to test this by having two kernels access a large array using strided accesses In one kernel, a gratuitous barrier is added into the loop, thus that threads in the workgroup are never more than one iteration away from each other. The speedup of the kernel with barriers over the kernel without barriers across all chips is shown in the *m-divg* row of Table X.

While all chips appear to benefit from the barrier, the clear outlier is MALI, on which adding the gratuitous barrier leads to a $6.45\times$ speedup. Thus, MALI appears to be extremely sensitive to intra-workgroup memory divergence – Figure 2 shows that sg is required for top speedups on MALI more than any other chip. Thus, while initially confounding, our findings suggests a new optimisation may be required to protect against memory divergence.

IX. Related and Future Work

a) Related Work : Our work aims to identify portable optimisations automatically. Prior work [8] has proposed statistical techniques for fixed environments, but our work allows reasoning over multiple dimensions – a necessity when dealing with graph algorithms. We cannot use randomisation a la STABILIZER since most OpenCL toolchains are proprietary. While it would improve the quality of the data we collect [10], our method would otherwise remain unchanged. Our method is motivated by prior work that shows empirical data from computer systems is largely not normally distributed, making techniques like ANOVA inapplicable, and instead uses quantile regression [9]. The MWU test is similarly non-parametric.

Muralidharan et al. [1] propose a technique for autotuning across architectures without retraining for the target architecture. Using performance data obtained from different Nvidia GPUs, and runtime data, an SVM is used to predict performant variants. Our work aims to construct *descriptive* models as opposed to predictive models, and treats GPUs, applications and inputs as black boxes. Appropriately suited to the the poor state of vendor OpenCL profilers, we require only the ability to time the execution of a program.

Varbanescu et al. [12] study the portability of three OpenCL graph algorithms using a CPU and two Nvidia GPUs and conclude that the effects of inputs swamped out any benefits gained by optimising them for specific OpenCL platforms. Our results support their findings that inputs play a significant role in performance, but we show that specialising for chips or applications is better than not optimising at all. Additionally, specialising across more than one dimension can deliver even more performance.

Merrill et al. [6] construct a performance-portable library of parallel primitives containing reductions, scans, sorting, etc. by encoding tunable parameters such as number of items per load, the number of threads per thread block, etc. in the CUDA/C++ type system. Their system is evaluated on three Nvidia GPUs and concludes similarly about the lack of a globally applicable optimisations for those problems. Other work [7], [32] has studied the performance portability of OpenCL on problems such as SGEMM, SpMV and FFT.

b) Future Work: The combination of increasing architectural diversity and magnitude-agnostic performance analysis provides many interesting avenues for future work. In particular, in this work we have used an *exhaustive* set of runtime results, across all application, input, and chip combinations. For future work, we want to explore whether smaller sample sizes from the test domain could be sufficient to yield significant results. This would not only cut down experimental time (allowing for larger domains), but also open up further applications, e.g. in developing *predictive* models, rather than the *descriptive* models shown in this work. Our models, in which architectural performance portability is a first class concern, are immediately applicable in compilers that target highly heterogeneous systems, such as TVM [33].

Finally, we aim to apply our magnitude-agnostic analysis technique to other domains, e.g. in dense linear algebra libraries. While prior work has examined this, we aim to explore if previous results were subject to bias or other shortcomings, as we've shown is possible in Section II. Such issues will become increasingly common as the diversity of architectures continues to grow and we need to use sound, automated techniques to debug issues in performance portability at scale.

X. Conclusion

We have shown that universally beneficial (or harmful) optimisations do not exist in the domain of graph algorithms. Therefore, we devised a data analysis that can consume experimental data and yield optimisation strategies tailored for portability by various degrees of specialisation; e.g. over chips, applications and inputs. Our analysis avoids reaching trivial or biased conclusions by using a magnitude-agnostic rank-based analysis. This analysis, run over a large study GPU graph algorithms on a diverse set of GPUs, allowed us to identify optimisation limits and quantify for the first time, the gap between specialisation and portability. By examining chip-specific optimisations, we were also able to identify several performance bottlenecks such as kernel launch latency, lack of atomic RMW combining, and memory divergence.

Acknowledgments

This work was supported by a Google Faculty Research Award and UK EPSRC IRIS Programme Grant (EP/R006865/1). We thank our anonymous reviewers whose comments improved the statistical presentation in this work.

REFERENCES

[1] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai, "Architecture-Adaptive Code Variant Tuning," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 325–338. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872411

[2] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning Algorithmic Choice for Input Sensitivity," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 379–390. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737969

[3] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A Framework for Adaptive Code Variant Tuning," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 501–512. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.59

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An Extensible Framework for Program Autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 303–316. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628092

[5] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542481

[6] D. Merrill, M. Garland, and A. Grimshaw, "Policy-based tuning for performance portability and library co-optimization," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[7] J. Price and S. McIntosh-Smith, "Exploiting auto-tuning to analyze and improve performance portability on many-core architectures," in *High Performance Computing*. Springer, 2017, pp. 538–556.

[8] C. Curtsinger and E. D. Berger, "STABILIZER: Statistically Sound Performance Evaluation," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 219–228. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451141

[9] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Why You Should Care About Quantile Regression," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 207–218. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451140

[10] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing Wrong Data Without Doing Anything Obviously Wrong!" in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508275

[11] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on GPUs," in *OOPSLA*, 2016, pp. 1–19.

[12] A. L. Varbanescu, M. Verstraaten, C. de Laat, A. Penders, A. Iosup, and H. Sips, "Can Portability Improve Performance?: An Empirical Study of Parallel Graph Analytics," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: ACM, 2015, pp. 277–287. [Online]. Available: http://doi.acm.org/10.1145/2668930.2688042

[13] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *PPoPP*, 2016, pp. 11:1–11:12.

[14] Nvidia, "nvGraph library user's guide," May 2018, https://docs.nvidia.com/cuda/pdf/nvGRAPH_Library.pdf.

[15] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. [Online]. Available: https://projecteuclid.org/euclid.aoms/1177730491

[16] Khronos OpenCL Working Group, "The OpenCL C specification," November 2013.

[17] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamaric, "Portable inter-workgroup barrier synchronisation for GPUs," in *OOPSLA*, 2016, pp. 39–58.

[18] K. Gupta, J. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *InPar*, 2012, pp. 1–14.

[19] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," in *IPDPS*, 2010, pp. 1–12.

[20] Khronos Group, "The OpenCL specification version: 2.0 (rev. 29)," July 2015, https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[21] Nvidia, "CUDA C programming guide, version 10.1," August 2019.

[22] D. Merrill, M. Garland, and A. S. Grimshaw, "High-performance and scalable GPU graph traversal," *TOPC*, vol. 1, no. 2, p. 14, 2015.

[23] Nvidia, "Parallel thread execution ISA: Version 6.0," Sept. 2017, http://docs.nvidia.com/cuda/pdf/ptx_isa_6.0.pdf.

[24] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically comparing memory consistency models," in *POPL*. ACM, 2017.

[25] B. Ashbaugh, "cl_intel_subgroups version 4," Aug. 2016, https://www.khronos.org/registry/OpenCL/extensions/intel/cl_intel_subgroups.txt.

[26] A. Lokhmotov, "ARM Midgard architecture," 2011, http://www.heterogeneouscompute.org/hipeac2011Presentations/OpenCL-Midgard.pdf.

[27] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, pp. 442–446.

[28] R. Jain, *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*, ser. Wiley professional computing. Wiley, 1991.

[29] T. Sorensen, "Inter-workgroup barrier synchronisation on graphics processing units," Ph.D. dissertation, Imperial College London, 2018, http://www.cs.princeton.edu/~ts20/files/phdthesis.pdf.

[30] K. O. McGraw and S. P. Wong, "A common language effect size statistic," *Psychological Bulletin*, vol. 111, no. 2, pp. 361–365, 1992.

[31] S. H. Lo, C. R. Lee, Q. L. Kao, I. H. Chung, and Y. C. Chung, "Improving GPU memory performance with artificial barrier synchronization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2342–2352, 2014.

[32] Y. Zhang, M. Sinclair, and A. A. Chien, "Improving Performance Portability in OpenCL Programs," in *Supercomputing*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jun. 2013, pp. 136–150. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-38750-0_11

[33] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: an automated end-to-end optimizing compiler for deep learning," in *Operating Systems Design and Implementation, OSDI*. USENIX Association, 2018, pp. 578–594.