Generating CUDA code at runtime: specializing accelerator code to runtime data

Tristan Perryman, Paul H. J. Kelly, Anton Lokhmotov and Tony Field
Department of Computing, Imperial College, London, United Kingdom
{p.kelly}@imperial.ac.uk

ABSTRACT

This abstract presents preliminary results from exploring the idea of generating code for a GPU accelerator at run-time. We show that this can lead to performance improvements due to specialization: we specialize the GPU code to the particular run-time data. We illustrate the prototype tool with a ray tracing example, where we achieve 10%–30% speedup due to specialisation to the scene being rendered. The prototype tool is based on our Taskgraph library for runtime code generation in C++; this supports runtime generation of both the CUDA kernel itself, and the host code for transferring parameters and results. The tool gives the programmer full, dynamic control over CUDA resource management and storage assignment.

1. INTRODUCTION

Programming heterogenous multicore systems, such as a PC with a GPU, is sometimes quite complex - so complex that it is easier to write a program to generate the implementation itself. This paper explores this idea. Our Taskgraph library (TGL)[1] supports runtime construction and execution of a subset of C at runtime. We have built a new back-end that generates CUDA instead of just C, and invokes the nVidia CUDA compiler, so that a C++ host program can build CUDA code on-the-fly.

In this paper we illustrate the use of the tool with a simple vector-scalar addition example, and present experimental results for a ray-tracing example where a performance benefit has been achieved by using the Taskgraph library to generate CUDA code specialised to data available only at run-time. This abstract is a brief summary of Perryman's thesis work [2], where more details are available. We plan to make the software publically available; please contact the authors for details.

2. THE TASKGRAPH LIBRARY

TGL was developed at Imperial College London to provide a flexible, but not-too-error-prone way to write programs that build C code at runtime, and then execute it. A minimal example is shown in Figure 2.

[1]URL: http://www.doc.ic.ac.uk/ phjk/Software/TGL/

Abstract submitted to *FPGAs, GPUs, and accelerators for High-Performance Computing Workshop* at Imperial College London, 17-18 September 2008

```
#include <TaskGraph>
using namespace tg;

int main() {
  int c = 1;
  TaskGraph < int, int, int > T;
  taskgraph( T, tuple2(x, y) ) {
    tReturn( x + y + c );
  }
  T.compile( tg::GCC );
  printf( "a+b+c = %d\n", T( 2, 3 ) ); // a+b+c = 6
}
```

Figure 1: TGL supports runtime contruction and specialisation of a subset of C from within a C++ program. In this example, `T` is the abstract syntax tree for a C function that takes two parameters, `x` and `y`, and returns `x+y+1`. The code is specialised to the value of `c` at construction-time.

TGL constructs an abstract syntax tree representation of the code (using the Stanford SUIF framework [1]). It then prints this to a temporary file, and calls the specified compiler. The resulting binary is dynamically linked into the client code so that it can be called. Parameter passing is type checked as with standard C.

3. USING TGL TO GENERATE CUDA

This section presents an example to show how the extended TGL for CUDA is used. The code constructs a vector in the host, and passes it to the GPU, where a scalar is added to each element. The result vector is then passed back to the host. See Figure 2. This simple example does not show the performance advantage of using TGL with CUDA; a more complex example is discussed in Section 4.

3.1 Vector-scalar addition in TGL+CUDA

Our philosophy in the design of TGL+CUDA was to give the programmer complete control over the CUDA that is generated. This is evident in Figure 4. This specifies how the work is mapped onto CUDA Cooperative Thread Arrays, and also explicit allocates memory and copies data to and from the host. Much of this is boilerplate which we plan to automate.

4. USING TGL+CUDA FOR SPECIALISATION: RAY TRACING

Although we have other TGL+CUDA examples [2], we focus here on the most complex and the nearest to a complete application.

```
int main( int argc, char *argv[] ) {
  cuda_host T;
  cuda_kernel T2;
  int k=5;
  int mem_size = sizeof(vector3)*NVALS;
  cudakerneltaskgraph(cuda_kernel, T2, tuple1(d_A))
  {
        ...See Figure 3...
  }
  cudahosttaskgraph( cuda_host, T, tuple1(h_A) )
  {
        ...See Figure 4...
  }
  T.compile(tg::NVCC, true, args);

  // Allocate and initialise vector operand
  vector3* h_A;
  h_A = (vector3*)malloc(mem_size);
  getVectors(h_A, NVALS);

  T(h_A); // invoke CUDA
}
```

Figure 2: Vector-scalar addition example - client code. The code elided here (and shown in Figures 3 and 4) creates Taskgraphs for the CUDA kernel, and also for the host CUDA code (which manages resources and parameter/result marshalling). Once this is done the GPU is invoked by a simple function call.

```
  cudakerneltaskgraph(cuda_kernel, T2, tuple1(d_A))
  {
    tVar(int, tx);

    bx = blockIdx.x;      // Block index
    tx = threadIdx.x;      // Thread index
    tVar(int, thisvector);
    thisvector = tx + bx*BLOCK_SIZE;

    d_A[thisvector].x = d_A[thisvector].x*k;
    d_A[thisvector].y = d_A[thisvector].y*k;
    d_A[thisvector].z = d_A[thisvector].z*k;
  }
```

Figure 3: TGL code to generate the CUDA kernel.

```
 cudahosttaskgraph( cuda_host, T, tuple1(h_A) ) {
   tInclude("stdio.h");   tInclude("cutil.h");
   tCall("CUT_DEVICE_INIT");
   // allocate device memory
   tVar(vector3*, d_A);
   tCall("cudaMalloc", cast<void**>(&d_A), mem_size);
   // copy host memory to device
   tCall("cudaMemcpy", d_A, h_A, mem_size,
         cudaMemcpyHostToDevice);
   // setup execution parameters
   tVar(dim3, threads);
   threads.x = BLOCK_SIZE;
   tVar(dim3, grid);
   grid.x = NVALS / threads.x;

   tCudaGlobalCall(T2, grid,threads,0,d_A);

   // copy result from device to host
   tCall("cudaMemcpy", h_A, d_A, sizeof(float3)*NVALS,
         cudaMemcpyDeviceToHost);
  tCall("cudaFree", d_A);
 }
```

Figure 4: TGL code to generate the host-side CUDA code.

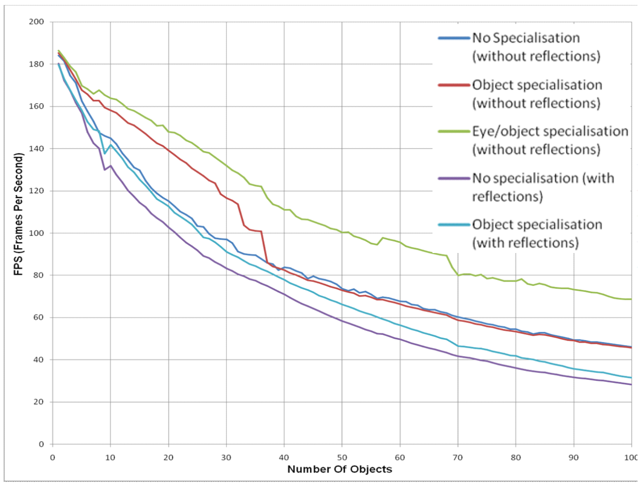

Figure 5: Performance of the specialised ray tracer. Maximum performance is achieved when specialising to both the scene's objects and viewpoint, without reflections (lighting remains variable). More realistic is to specialise to objects but not viewpoint (and with reflections); here speedup is about 10%, giving 30 frames/second for a scene with 100 objects.

We started with a simple CUDA ray tracer written by Eric Rollins[2]. This is a very basic ray tracer, computing diffusion and specularity in a scene consisting of spheres. We extended it to handle multiple light sources, shadows and multiple reflections, and improved its performance using constant memory instead of shared memory for the scene data. Figure 5 shows the performance, over this improved baseline, gained from using TGL+CUDA in a simple real-time ray tracing application. Compilation times (once for each scene) in this example range from about 1.8 to about 11 seconds, depending on the amount of loop unrolling done. The graph shows a drop in performance in the without-reflections case which we have not yet fully understood. If we limit unrolling to the first 30 objects the effect disappears and some of the benefit of specialisation is seen in larger scenes. (Configuration: 2.8ghz E6300 Core2Duo with 2GB of 800MHz DDR2 RAM, Nvidia 8800GTS with 640MB video memory. Open SuSE Linux 10.2, Nvidia driver 171.06, Cuda Toolkit and SDK 1.1).

5. CONCLUSIONS

TGL+CUDA supports specialisation which can yield performance benefits on GPUs. Our future work will focus on using it to ease generation of more sophisticated implementations.

Acknowledgements. This work was partially funded by the U.K. Engineering and Physical Sciences Research Council through the SPOCS project (ref EP/E002412).

6. REFERENCES

[1] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.

[2] T. Perryman. Runtime compilation with nvidia cuda as a programming tool. Technical report, Imperial College London, 2008. BSci dissertation.

[2]URL: http://eric_rollins.home.mindspring.com/ray/cuda.html