Helmholtz Challenge

Oskar Weigl (ow610)
Ryan Savitski (rs5010)
Yong Wen Chua (ywc110)

1 Introduction

This report outlines the acceleration of the given FEM code using a GPU. The hardware architecture is a natural fit for the target problem, which exhibits significant data parallelism with latency tolerance towards individual elements of the iteration space. The application is compute-bound through floating point computations, which GPU architectures are designed to perform efficiently. Furthermore, there is no interleaving of IO or serial kernels, avoiding the need of copying any intermediate results between host and device.

2 Initial investigation

2.1 Default implementation

The computation is performed on triangular prisms with six edge nodes obtained from an extruded unstructured mesh. The vertical columns of nodes are a regular structure and therefore allow for the data to be laid out sequentially for each column, reducing the number of indirect memory accesses.

The given implementation consists of a serial execution of the following stages:

`expression 1` sets up the boundary conditions. Note that the autogenerated is particularly inefficient, initialising nodes by iterating over cells and computing the same result multiple times. This can easily be reimplemented as a serial loop over the coordinate array.

`lhs and rhs` kernels perform local assembly with eight double precision multiply-accumulate expressions per node of a cell (arranged in a reduction tree), iterating over cells in column-first order. Note that the two sides of the equation are independent and could be computed in parallel on a device with sufficient resources.

The provided implementation, processing the `small` mesh on a i7-3770K CPU (see appendix A.1 for more details) takes (averaged over a set of runs) 69.18s (boundary: 6.90s, interpolation: 2.58s, RHS: 13.4s, LHS: 46.3s). Dynamic voltage and frequency scaling (DVFS) was enabled, making the frequency of the relevant core float between 3.5 GHz and 3.9 GHz during execution.

2.2 Initial look at parallelisation

Looking at serial runtimes, we see that `lhs` and `rhs` kernel calls are the most expensive. Boundary condition generation (`expression 1`) is trivial to speed up on the GPU (see section 3) and does not require parallelisation. Parallelising `lhs` and `rhs` kernels however is going to directly improve the speed of the overall computation.

It can easily be seen that all of the cell computations are independent, given that the global vector accumulation is made atomic. Possible iteration space parallelisation strategies include parallelising over either cells, columns of cells, the `ip` loop or some combination of these.

The initial parallelisation of the code was done on the CPU using Intel Threading Building Blocks (TBB)[1]. TBB is a C++ template library for expressing task-based parallelism and includes dynamic task scheduling and scalable memory allocators. The library does not require a special compiler, can be used productively and generates performance-portable code across processor with widely different core counts.

The TBB version of the FEM code was written to ensure that the chosen parallelisations are correct and in general was very straightforward, essentially replacing `for` loops with `parallel_for` constructs.

2.3 Timing code changes

Given that the target device for final optimisation was an Nvidia GPU, with the intention to use CUDA language and tooling, the operating system was chosen to be Windows for profiling tool support. As a consequence, timing code had to be ported, instead using the Windows API function `QueryPerformanceCounter`[2] which provides time stamps with 1 microsecond resolution. Its implementation is opaque, but guaranteed to be accurate, consistent and monotonic. It is likely using the invariant Time Stamp Counter (TSC) found on the CPU, with necessary OS-level compensation, making timestamps synchronised across processor cores and independent of DVFS.

3 GPU and CUDA

3.1 GPU hardware

The target device is an Nvidia GTX 670 GPU with Kepler GK-104 architecture. The architecture is based on 6 Streaming Multiprocessors (SMX). Each SMX consists of an instruction cache, four warp schedulers with two dispatch units each, a 64k entry 32-bit register file, L1 cache and shared memory [3]. The functional units are grouped into 15 functional unit blocks, with varying number of SIMD lanes [4]. A breakdown of functional unit counts per SMX is shown in table 1. The CUDA cores are the main functional units, which can execute a single precision floating point instruction per clock cycle. Five out of six blocks of the CUDA cores can also execute basic 32 bit integer arithmetic instructions.

This architecture has a fundamental Single Instruction Multiple Thread (SIMT) width of 32, referred to as a warp. Each SMX has 4 warp shedulers, each managing up to 16 active warps. Each clock cycle, each warp sheduler selects a warp from the pool of warps ready to execute. Each warp scheduler is capable of super-scalar issue of 2 instructions per cycle from a selected warp. Therefore, if there is sufficient Instruction Level Parallelism (ILP) in the currently executing context and no structrual hazards, the collection of warp schedulers can issue up to 8 warp-wide instructions per clock cycle.

3.2 Initial GPU implementation

The starting point for the GPU version was a straightforward CUDA rewrite of kernels as follows:

As discussed above, the default boundary condition setting `expression 1` kernel was inefficiently iterating over cells to initalise the node array, overwriting adjacent cells' results and therefore doing a factor of roughly 36 redundant work for the given small mesh. The CUDA kernel version iterates directly over the array (each iteration packed into a separate thread). It is worth noting

Table 1: GK-104 functional units per SMX [4].

width	Type
32	CUDA cores (#1)
32	CUDA cores (#2)
32	CUDA cores (#3)
32	CUDA cores (#4)
32	CUDA cores (#5)
32	CUDA cores (#6)
16	Load/Store Units (#1)
16	Load/Store Units (#2)
16	Interpolation SFUs (#1)
16	Interpolation SFUs (#2)
16	Special Function SFUs (#1)
16	Special Function SFUs (#2)
8	Texture Units (#1)
8	Texture Units (#2)
8	CUDA FP64 cores

that the boundary code uses cosines, which are accelerated in hardware on the GPU and thus are significantly faster.

The `zero` kernel was turned into a `cudaMemset`, executing in a millisecond.

The `expression 2` kernel was observed to be doing a trivial direct copy operation, which is removed by simply passing the original buffer pointer to the next kernel.

For `rhs` and `lhs` kernels, each thread performs the whole computation over a cell (i.e. the 8x6x6 nested loop). The threads are packed into blocks in column order, to allow adjacent threads in a warp to do coalesced reads and writes to input/output data as it is stored sequentially for a column of nodes. As in the TBB version, accumulation of results to the global vector was made atomic.

To verify correctness, buffer states after each kernel pass were copied to the host and compared against the results of its reference computation. Naturally, for release builds, all memory is kept on the device until the final output buffer copy back.

The straightforward CUDA implementation performed the same computation in 5.68 seconds on average, a 96% increase in performance over the serial version on the CPU, which is explained by the natural fit of the GPU for the problem.

3.3 Constant memory optimisation

The direct implementation saw a high miss rate in L1 and L2 caches (see figure 1), with each miss having to go to off-chip memory which was detrimental to performance. This is a consequence of storing the constant factor matrices on the local stack. Since these factors are constant across all cells for a given kernel, the obvious optimisation is to push this data into constant memory, since it "can be as fast as a register"[6] if all threads access the same location. Furthermore, there is an additional benefit as the GK-104 architecture has a separate constant memory cache per SMX that is big enough to cache all constant matrices.. The effects of this optimisation on

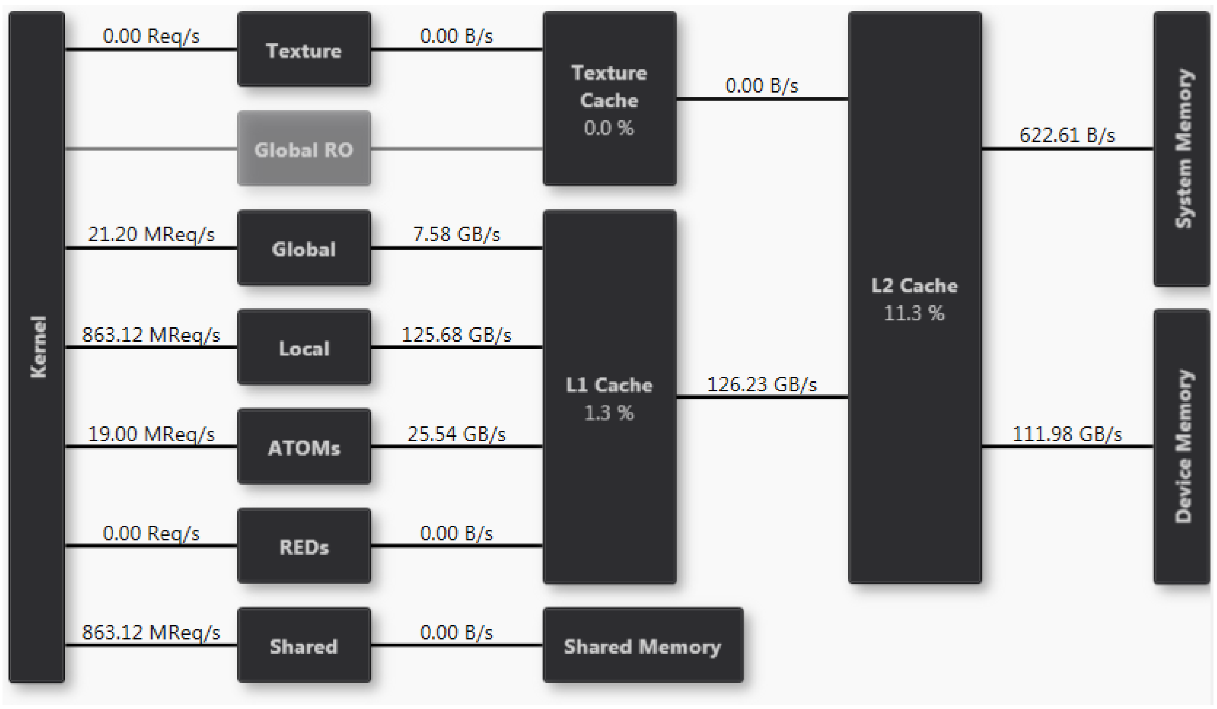

Figure 1: Memory bandwidth overview of original implementation.

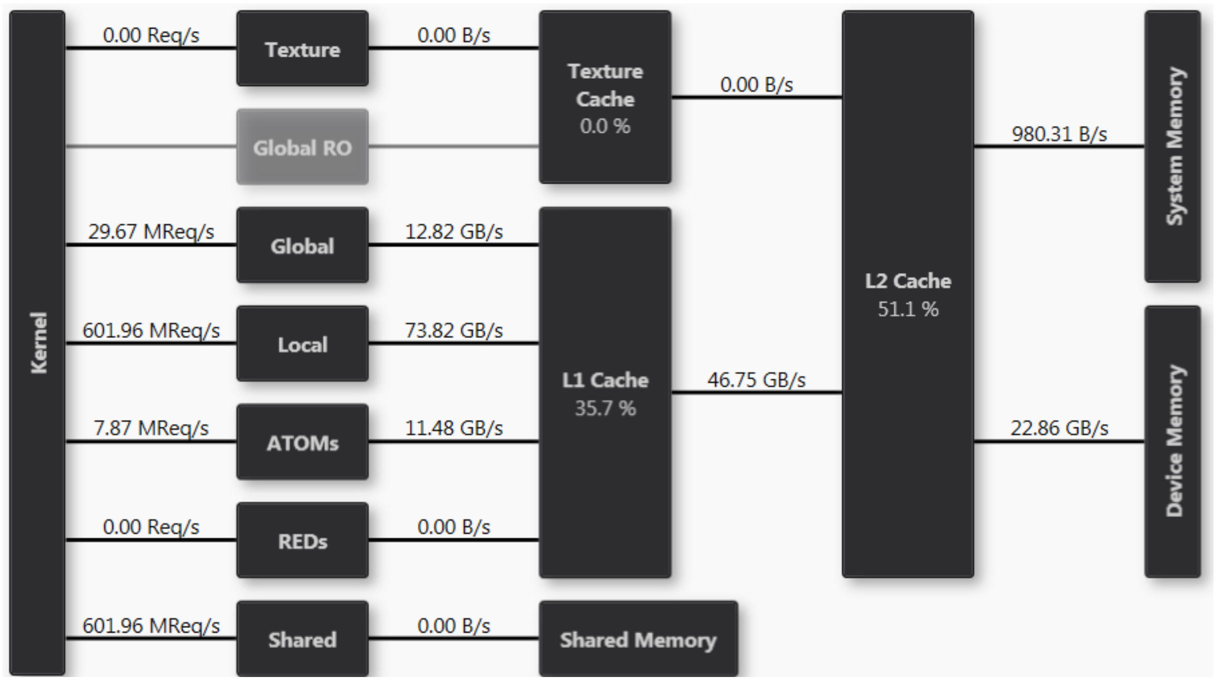

Figure 2: Memory bandwidth overview of constant memory implementation.

Table 2: Comparison of execution times for constant memory optimisation on small mesh.

	Execution time (s)		Local Memory Use	
Memory	**Local**	**Constant**	**Local**	**Constant**
Kernel				
LHS	3.08	1.28	385GB	93GB
Total	5.68	2.58		

`lhs` and total execution time can be seen in table 2. Figure 2 shows the cache hit rates and memory bandwidth for the optimised version.

3.4 Reducing register spill

The `lhs` and `rhs` kernels are expensive in computation and contribute to the majority of the execution time. This section looks at further optimisations applied to the `lhs`, with the `rhs` benefitting from the same optimisations as the kernels are very similar.

The `lhs` kernel still suffers from significant local memory traffic, originating from register spilling. The causes are investigated below.

During the hot loop of the kernel, the original implementation accumulates the result of the main expression to 36 distinct accumulators. This is later summed to 6 accumulators, which are then atomically added to the nodes stored in global memory. As the compiler cannot treat floating point addition as associative, it cannot do the 36 to 6 reduction early. Therefore the state of all 36 accumulators is retained throughout the loop. As we use double precision, this would require 72 32-bit registers, while the GK-104 has a maximum register allocation of 63 32-bit registers per thread [5].

A straightforward optimisation attempted was to rewrite the code to only use the 6 required accumulators, reducing the number of registers used to 58, and producing exactly zero local memory usage. However, the performance is lower than than the original implementation that spills registers.

After experimenting with the number of accumulators and interchange of the `ip j k` loops, we found the structure shown in listing 1 to be optimal. This structure uses 12 accumulators, and spreads out the distance between accesses to the same accumulator as much as possible. Checking the disassembly, the compiler unrolls all three levels of this loop. A good guess for why the presented structure is optimal is that the compiler will interleave ajacent iterations of the loop if they are independent. Apparently an independence distance of 6 iterations is not enough, but 12 is sufficient to remove dependency stalls.

The execution time improvements are presented in table 3.

Listing 1: LHS kernel main loop

```
for (int ip = 0; ip<8; ip++){
  for (int k = 0; k<6; k++){
    for (int j = 0; j<6; j++){
      A[k%2][j] += LargeExpression(ip,j,k);
    }
  }
}
```

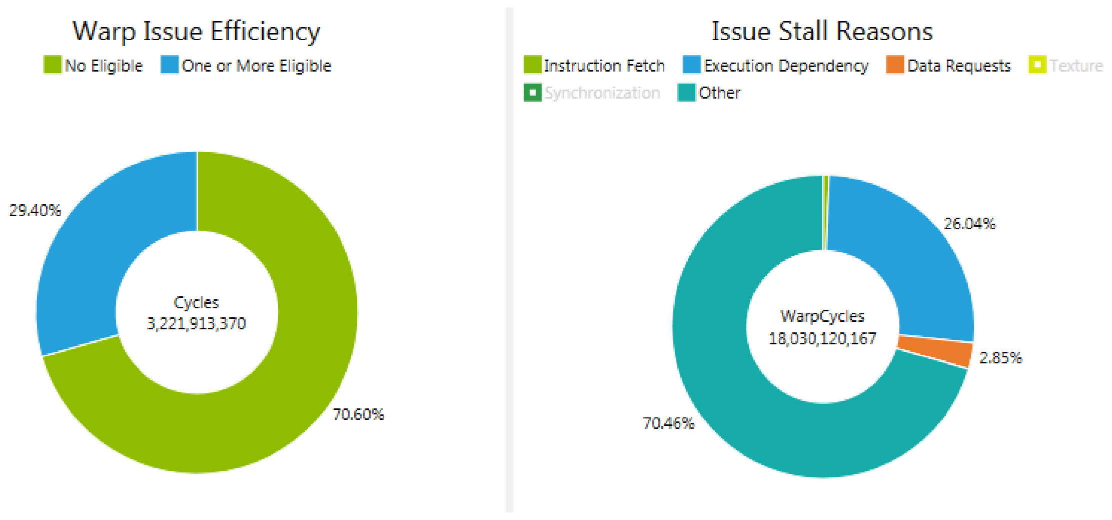

Figure 3: `lhs` issue efficency and stall reasons. Note that average serialisation was 57%.

Table 3: Execution times before and after optimisations

Kernel	Before	After
expr_1	27ms	26ms
rhs_1	158ms	158ms
rhs	2.29s	714ms
lhs	3.10s	980ms

3.5 Final Perfomance Remarks

As presented in Section 3.1, we know that we only have eight double precision units per SMX. This ends up being the ultimate bottleneck for performance on this device. As we have four warp schedulers trying to schedule onto only a single double precison functional unit, we should, at least for double precision arithmetic instructions, expect a 25% issue efficiency. Presented in Figure 3 is the issue efficiency and the stall reasons for the final `lhs` kernel implementation. We see that the category "other" is by far the biggest stall reason, which is contributed to by functional units being busy in a given cycle. This means that we are very close to the fundamental limit of double precision execution for this device.

The profiling tools report an instruction serialisation metric of 57%. This is lower than the theoretical 75% from eight double precision units across 32 threads, with the difference being accounted for by interleaving of other types of instructions.

An interesting consequence of this bottleneck and a small benefit is that the thermal output of the device is less than 75% of TDP. This means that DVFS brings the GPU to its maximum frequency during the entire execution.

Finally, both the analysis of the archtecture and profiling results suggest that if double precision is nessecary for the application, then perhaps the GK-104 GPUs are not the best choice. Indeed, the successor GK-110 has 64 double precision units per SMX, and would be a much better choice for this workload.

Theoretically, double precision can be emulated through two floats, however the implementations are roughly 8 times slower than single precision and thus would not be worthwhile for this architecture due to additional register pressure. Alternatively, mixed precision approach to the algorithm could be employed if double precision is not crucial for all stages of the computation.

The final execution time for the total computation on the GPU is on average 2.28s, a 97% speedup over the initial code on the CPU.

4 Work partition

Yong Wen Chua	- TBB, CUDA.
Oskar Weigl	- CUDA, profiling.
Ryan Savitski	- code analysis.

Appendices

A Summary and setup

A.1 CPU

```
Processor 1     ID = 0
  Number of cores   4 (max 8)
  Number of threads 8 (max 16)
  Name      Intel Core i7 3770K
  Codename    Ivy Bridge
  Specification   Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz
  Package (platform ID) Socket 1155 LGA (0x1)
  CPUID     6.A.9
  Extended CPUID    6.3A
  Core Stepping   E1/L1
  Technology    22 nm
  TDP Limit   77 Watts
  Tjmax     105.0 C
  Core Speed    2807.4 MHz
  Multiplier x Bus Speed  28.0 x 100.3 MHz
  Stock frequency   3500 MHz
  Instructions sets MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX
  L1 Data cache   4 x 32 KBytes, 8-way set associative, 64-byte line size
  L1 Instruction cache  4 x 32 KBytes, 8-way set associative, 64-byte line size
  L2 cache    4 x 256 KBytes, 8-way set associative, 64-byte line size
  L3 cache    8 MBytes, 16-way set associative, 64-byte line size
  FID/VID Control   yes

  Turbo Mode    supported, enabled
  Max non-turbo ratio 35x
  Max turbo ratio   39x
  Max efficiency ratio  16x
  Min Power   60 Watts
  O/C bins    unlimited
  Ratio 1 core    39x
  Ratio 2 cores   39x
  Ratio 3 cores   38x
  Ratio 4 cores   37x
  TSC       3510.2 MHz
  APERF     3730.9 MHz
  MPERF     3481.3 MHz
```

A.2 GPU

```
Display adapter 0
  Name      NVIDIA GeForce GTX 670
  Revision    A2
  Codename    GK104
  Technology    28 nm
  Memory size   2 GB
  PCI device    bus 1 (0x1), device 0 (0x0), function 0 (0x0)
  Vendor ID   0x10DE (0x3842)
  Model ID    0x1189 (0x2678)
  Performance Level 0
    Core clock  1175.0 MHz
    Memory clock  3105.0 MHz

Win32_VideoController   AdapterRAM = 0x80000000 (2147483648)
Win32_VideoController   DriverVersion = 9.18.13.3523
Win32_VideoController   DriverDate = 03/04/2014
```

A.3 Operating system

```
Windows Version     Microsoft Windows 7 (6.1)
                    Ultimate Edition 64-bit  Service Pack 1 (Build 7601)
```

A.4 Parallelisation paradigms

Intel Threading Building Blocks on the host CPU during initial investigations. CUDA on GPU.

A.5 CUDA

Compiler:

```
nvcc: NVIDIA (R) Cuda compiler driver
Built on Wed_Jul_10_13:36:45_PDT_2013
Cuda compilation tools, release 5.5, V5.5.0
```

Compilation flags:

```
nvcc.exe -gencode=arch=compute_30,code=\"sm_30,compute_30\" --use-local-env
         --cl-version 2012 -maxrregcount=0  --machine 64 --compile -cudart static
         -DWIN64 -DNDEBUG -D_CONSOLE -D_MBCS -Xcompiler "/EHsc /W3 /nologo /O2 /Zi /MD"
```

References

[1] Intel, *Intel Threding Building Blocks.*
`https://www.threadingbuildingblocks.org`

[2] Microsoft, *Developer reference - QueryPerformanceCounter function.*
`http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85)`
`.aspx`

[3] Nvidia, *Whitepaper - NVIDIA GeForce GTX 680.*
`http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.`
`pdf`

[4] Ryan Smith, *The Kepler Architecture: Fermi Distilled.*
`http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review/2`

[5] Nvidia, *Cuda-C Programming Guide.*
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

[6] Nvidia, *CUDA-C Best Practices Guide.*
`http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`