Model-driven optimisation of memory hierarchy and multithreading on GPUs

Andrew A. Haigh **Eric C. McCreath**

Research School of Computer Science,
The Australian National University,
Canberra, Australia
Email: {andrew.haigh,eric.mccreath}@anu.edu.au

Abstract

Due to their potentially high peak performance and energy efficiency, GPUs are increasingly popular for scientific computations. However, the complexity of the architecture makes it difficult to write code that achieves high performance.

Two of the most important factors in achieving high performance are the usage of the GPU memory hierarchy and the way in which work is mapped to threads and blocks. The dominant frameworks for GPU computing, CUDA and OpenCL, leave these decisions largely to the programmer. In this work, we address this in part by proposing a technique that simultaneously manages use of the GPU low-latency shared memory and chooses the granularity with which to divide the work (block size).

We show that a relatively simple heuristic based on an abstraction of the GPU architecture is able to make these decisions and achieve average performance within 17% of an optimal configuration on an NVIDIA Tesla K20.

Keywords: Parallel programming, Graphics Processing Unit (GPU), optimisation, performance analysis.

1 Introduction

It is well known that writing code that gives high performance on GPUs is a challenging task. This is due to a number of factors, including the complex memory hierarchy and multiple levels of parallelism. Automated optimisation is difficult due to the existence of local optima in the optimisation space (Zhang & Mueller 2012), and challenges in predicting the effect of combining optimisations (Ryoo et al. 2008).

In this paper, we investigate the problem of simultaneously balancing the GPU's occupancy and resource usage in order to achieve high performance. This is done by using a cost model that can take into account the balance between these factors. This is done by static analysis of the source code and device to extract features of the computation and hardware. Then, we construct a model that is evaluated exhaustively for all possible variants of the kernel in order to determine the best configuration.

On a NVIDIA Tesla K20 this technique is shown to achieve performance on average within 17% of an optimal configuration. This can be achieved an order of magnitude faster than an empirical search. In addition, it does not require usage of the GPU itself to perform this search.

We present an overview of related work followed by a description and evaluation of this technique.

1.1 GPU architecture

The graphics processing unit (GPU) is an architecture suited to highly parallel computations. It operates in a SIMD fashion, i.e., the user writes a kernel that runs the same computation on many different pieces of data simultaneously. The hardware is subdivided logically into a number of symmetric multiprocessors (SMs), each of which contains a number of cores and hardware instruction schedulers. Each SM is allocated a number of blocks (work-groups) that consists a group of threads.

The architecture is throughput-oriented such that if one thread is waiting due to a high-latency instruction or dependency, another can execute during the intervening cycles.

Each SM contains a fixed amount of block-private shared memory, which is available to the threads allocated to that SM. Due to the throughput-oriented nature of GPUs, the cache is useful primarily for exploiting spatial locality in data, and not temporal locality. Hence, the shared memory is typically used to retain data that has to be accessed multiple times over the lifetime of the kernel. Shared memory usage is controlled by the programmer in the popular CUDA and OpenCL frameworks.

A number of authors (Ryoo et al. 2008, Daga et al. 2011) give a classification of the kinds of optimisations that are useful for achieving high performance in GPUs. Broadly, these are, utilise occupancy (cover latency with parallelism), utilise memory hierarchy, data layout, minimise divergence, and minimise instruction count. In this work, we consider how to optimise for the first two factors.

1.2 Related work

There are a number of existing frameworks that allow the programmer to manage shared memory from a high level perspective. CUDA-lite (Ueng et al. 2008) is a general framework for CUDA codes that uses annotations to allow the programmer to specify transformations that take a kernel accessing only global memory and modify it to use shared memory, correctly inserting the code to load/store the relevant values. Other frameworks, such as hiCUDA (Han & Abdelrahman 2009), are able to generate GPU code

from high-level annotations of serial code. However, these frameworks still leave the burden of choosing the right transformations to achieve high performance to the user.

Early literature on general optimisation principles for GPUs identifies the issue of achieving a balance between resource usage and active threads (Ryoo et al. 2008) in a qualitative sense.

There is a substantial body of literature on techniques for auto-tuning GPU code. A large number of works have focused on tuning specific algorithms, such as stencil computations (Zhang & Mueller 2012, Datta et al. 2008, Kamil et al. 2010, Holewinski et al. 2012). All of these works propose exhaustive empirical search of the parameter space, which may be unsuitable for problems tuning a large number of parameters.

A number of authors propose methods for searching the auto-tuning space. Bergstra et al. (2012) use a machine learning technique to learn a model that can be used to predict runtimes for the purpose of choosing block size and memory layout for a stencil code. Ma & Agrawal (2010) propose a framework for optimising shared memory usage using an integer linear programming formulation. However, their cost model does not take into account the effect of changing the occupancy/block size. Instead, they perform this search empirically.

Baskaran et al. (2008) focus on parallelising loops with an emphasis on careful usage of the memory hierarchy. However, they choose to place all reusable data into shared memory, and introduce an extra level of tiling that can be used to reduce the working set if it is too large to fit into the shared memory.

Complementary work on utilising a small scratchpad memory has been studied in other domains such as embedded systems (Avissar et al. 2002).

2 Approach

Our aim is to simultaneously manage the occupancy and use of shared memory for a given kernel, with the goal of optimising for maximum performance.

It is well established (Ma & Agrawal 2010) that in general placing reusable data in shared memory improves performance because it reduces the number of off-chip high-latency global memory accesses. However, because the shared memory must be shared between all active blocks on an SM, increasing the shared memory usage per block may decrease the number of blocks (and hence threads) that can be 'in-flight' at any given time. This may worsen performance if the number of threads is not enough to cover these high-latency operations and other dependencies. Thus, we intend to explore this trade-off.

The computation is divided into blocks. We denote N as the number of threads in each block, and F determines the number of computations mapped to each thread. Thus, each block is responsible for NF independent calculations scheduled to the threads in a round-robin fashion.

The rationale for allowing $F > 1$ is that it may improve performance by amortising the total cost of instructions that are common to all threads and providing more independent instructions to exploit instruction level parallelism (ILP) within a thread without stalling the pipeline.

In addition, we allow the choice of which of the arrays accessed by the kernel to cache in shared memory. If the array is cached, the region of the array that is required to be read will be preloaded by a block cooperatively before it is needed and, if modified, written back to global memory at the kernel's completion. This avoids having to potentially fetch the same data multiple times from global memory.

2.1 Performance model

Our approach relies on a heuristic to estimate the performance of each possible configuration for a given kernel. Our heuristic can be evaluated without compiling the kernel. The number of configurations is small enough that we can estimate the best configuration by applying the heuristic to each. Performance models of GPU execution have been proposed by a number of authors, the most notable being Hong & Kim (2009).

Occupancy. A preliminary to the full model is to determine statically the number of blocks that can be active simultaneously on a single SM. Hardware limits dictate that the number of blocks that can run simultaneously on a single SM is given by the largest k s.t.

$$k = \min\left(\text{max blocks/SM}, \frac{\text{shared memory/SM}}{\text{shared memory/block}}, \frac{\text{max threads/SM}}{N}, \frac{\text{registers/SM}}{N(\text{registers/thread})}\right)$$

All of our benchmarks consist of at least one point in the code where every variable in shared memory is active. Thus, the shared memory usage per block is given simple by the sum of all the allocations that we make. We have observed that register usage is not a limiting factor for any of our benchmarks. Due to the difficulty in estimating it, we have ignored it here.

Memory access cost. This is an attempt to estimate the time required to make all the read/writes to the high-latency global memory, which is bounded by the total global memory bandwidth. It is given by

$$\text{memory cost} = \frac{\text{total reads/writes}}{\text{effective bandwidth}}$$

We ignore the effect of L1/L2 cache hits/misses due to the fact that GPU caches are primarily optimised for spatial rather than temporal locality. The total number of reads/writes are easily counted via inspection of the kernel, under our framework described below. We distinguish between coalesced and non-coalesced accesses and adjust the cost accordingly (non-coalesced accesses require potentially transfer of a whole cache line). The maximum bandwidth can be computed from specifications available by probing the device. However, this is only achieved if the number of threads is enough to provide work to do during the period of memory access latency (Hong & Kim 2009). The detailed estimation of achieved bandwidth is given in Figure 1.

Computation cost. Though our kernels, if properly implemented, should be memory-bound, a poor configuration (i.e. with very low occupancy or superfluous shared memory use) may be instruction-throughput bound. Thus, we also factor the cost of arithmetic operations into the model to identify bad configurations.

Our framework allows counting the number of assignment operations. Inspection (using `cuobjdump`) of the binaries shows around 5 instructions for each calculation, taking into account load/store instructions and calculating addresses. We add this to the total cost using a calculation analogous to above that factors in how many threads are required to saturate the arithmetic pipeline.

$$\text{saturating threads} = \frac{\text{maximum bandwidth}}{\text{number of SMs}} \frac{\text{memory latency}}{\text{core clock}}$$

$$\text{effective bandwidth} = (\text{maximum bandwidth}) \min\left(1, \frac{4NFk}{\text{saturating threads}}\right)$$

Figure 1: Detailed equation for memory throughput

Table 1: Specifications of GPUs used in tests

	Device	
Specifications	GTX580	K20
SMs	16	13
Clock speed (MHz)	1564	705
Memory bandwidth (GB/s)	192.4	208
Memory latency (cycles)	400-600	200-400
Cores/SM	32	192
(4 byte) Registers/SM	32k	65k
Sh. Mem/SM (kB)	48	48
Max threads/SM	1536	2048
Max blocks/SM	8	16

Synchronisation. In order to model the effect of block synchronisations that are necessary when sharing data between threads, we multiply the estimated runtime by 2 for configurations where only one block can run per SM and the inner loop of the kernel contains a block synchronisation. This is to approximate the fact that under such circumstance we only have on average half the number of active threads.

3 Results

We restrict attention to the case in which the work to be done by the kernel consists of M independent computations. Accordingly, the basic kernel does not contain any block synchronisations and we do not have to consider issues of correctness with regard to mapping work to blocks.

To implement our approach we use benchmarks that are templated with N and F are template parameters. This avoids any overhead (e.g. not loop unrolling) of having these values unknown at compile time.

The benchmarks are written in a very small subset of CUDA C such that it is possible to extract relevant features of the kernel using a preprocessing script. Loops are annotated with a fixed trip count to facilitate statically determining properties of the kernel. We provide a wrapper library written in Python that is able to both analyse, and compile and run a particular configuration governed by the parameters above.

To implement the shared memory management, all kernels are originally written making only use of global memory accesses. They are annotated with directives that indicate at which point to load/store an array into shared memory and the region of the array that will be accessed by the kernel. If shared memory use is enabled for an array, any intermediate accesses are remapped from global memory to shared memory and the directives are replaced with the correct code to preload/store the data into shared memory.

We evaluate this approach on the following simple benchmark kernels that have patterns of computation similar to some members of the Rodinia benchmark suite (Che et al. 2009) used in previous GPU literature.

- `increment`: a kernel that reads an array and increments each element by one. This is purely a test of whether the chosen configuration achieves a high memory throughput.
- `dense`: a dense matrix-vector product.
- `dense2`: dense matrix-vector product, but with the matrix stored in row-major order, so that accesses to it are non-coalesced, unless a block is prestored to shared memory.
- `kmeans`: calculating the distance between all pairs of points in two sets.
- `blur`: a synthetic 1D stencil that exhibits potential data reuse between threads.

For each benchmark, we compare the real runtime of the configuration estimated by the heuristic (*heuristic*) to have the best performance to the runtime of the best configuration (*optimal*) found by exhaustively searching the optimisation space of possible combinations of block sizes and shared memory mappings. For all tests, we ensure that M is large enough so that for all choices of N and F we create enough blocks to schedule to all SMs to avoid load imbalancing. For the `dense` benchmark on the K20, we use a larger data set compared to the GTX580, because more memory allows a higher M, which reduces this effect.

The relative performance of configurations is likely to vary significantly depending on the underlying architecture, especially given that our heuristic tunes for a particular device. Therefore, we apply this methodology to two different GPUs, an NVIDIA GTX 580 and a NVIDIA Tesla K20. Table 1 gives the relevant specifications for each of these devices. They differ significantly in a number of factors that are key in the behaviour of the benchmarks that we consider. Thus, we consider this a good test of the generality of our approach.

Table 2 gives for each benchmark and device the runtime for the configuration chosen by our heuristic and the best found runtime. Each result is averaged over 10 runs of each kernel. To demonstrate how our transformations help to approach theoretical limits on performance, we also compare (except for `kmeans`, which, if all data is cached, is not memory bound) to the lower bound which is the time required to make all the transfers between the SMs and global memory assuming peak memory bandwidth, which is similar for both devices.

The average slowdown compared to the optimal configuration for K20 across all benchmarks is around 17%. It takes less than 30 seconds to choose the best configuration according to the heuristic, compared to on average two hours to search all configurations empirically. Thus, the overhead of evaluating our heuristic is very small.

We note the fact that the heuristic is able to somewhat successfully choose configurations suited to individual devices. Qualitative analysis of the chosen configurations run on the other device showed relatively poor performance in a number of cases.

We note the relatively poor result for the K20, which has much higher peak performance than the GTX580. This is largely due to the fact that the kernels are memory bound and it is not possible to take advantage of this extra power. In addition, our framework does not attempt to make use of the larger caches and register files present on this architecture.

We also note that successes in having any configuration (not necessarily the one picked by our heuristic) with high performance is evidence that the set of

Table 2: Runtimes (ms) for each benchmark under different configuration selection policies. Slowdown is for heuristically chosen configuration relative to optimal configuration.

Device	Total	Lower	GTX580			K20		
Policy	Configs	bound	Optimal	Heuristic	Slowdown	Optimal	Heuristic	Slowdown
`increment`	96	0.044	0.054	0.056	3.3%	0.054	0.056	4.2%
`dense`	192	5.58	6.25	6.27	0.3%	17.8	17.9	0.2%
`dense2`	192	5.58	30.7	46.0	49.9%	43.1	51.0	18.4%
`kmeans`	768	n/a	34.1	47.2	38.4%	28.9	41.8	44.6%
`blur`	576	4.19	5.54	9.36	68.8%	6.44	7.68	19.3%

transformations we consider, particularly the aggressive use of F, are useful as a general strategy for writing high performance kernels. In particular, $F = 2$ or 4 could be combined with the use of vectorised instructions present on some GPUs.

4 Conclusion

We aim to address the problem of simultaneously managing occupancy and reuse of data via shared memory, two of the most important considerations when writing efficient code for GPUs. We have demonstrated that it is possible to use a simple cost model to determine the balance between these factors and achieve high performance. Our results show that we achieve on average within 17% of the best found configuration on a Tesla K20.

At present, our framework can only handle computational patterns that are essentially one dimensional. We intend to extend our approach to 2D and 3D patterns and also extend our approach to handle more complicated compositions of loop nests. Subsequently, we intend to evaluate our approach on some real-world examples.

Our performance model does not consider any caching effects, which is reflected in the weaker result for kernels with more complicated memory access patterns. This will be increasingly important in the future as newer GPUs contain more sophisticated caches to enable the transition from being purely throughput-oriented to being able to handle more diverse workloads. In particular, the resulting problem of predicting inter-thread and inter-block caching effects is not well understood.

References

Avissar, O., Barua, R. & Stewart, D. (2002), 'An optimal memory allocation scheme for scratch-pad-based embedded systems', *ACM Transactions on Embedded Computing Systems (TECS)* **1**(1), 6–26.

Baskaran, M. M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A. & Sadayappan, P. (2008), Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories, *in* 'Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming', ACM, pp. 1–10.

Bergstra, J., Pinto, N. & Cox, D. (2012), Machine learning for predictive auto-tuning with boosted regression trees, *in* 'Innovative Parallel Computing (InPar), 2012', IEEE, pp. 1–9.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. & Skadron, K. (2009), Rodinia: A benchmark suite for heterogeneous computing, *in* 'Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on', IEEE, pp. 44–54.

Daga, M., Scogland, T. & Feng, W. (2011), Architecture-aware mapping and optimization on a 1600-core GPU, *in* 'Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on', IEEE, pp. 316–323.

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J. & Yelick, K. (2008), Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, *in* 'Proceedings of the 2008 ACM/IEEE conference on Supercomputing', IEEE Press, p. 4.

Han, T. D. & Abdelrahman, T. S. (2009), hiCUDA: a high-level directive-based language for GPU programming, *in* 'Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units', ACM, pp. 52–61.

Holewinski, J., Pouchet, L. & Sadayappan, P. (2012), High-performance code generation for stencil computations on GPU architectures, *in* 'Proceedings of the 26th ACM international conference on Supercomputing', ACM, pp. 311–320.

Hong, S. & Kim, H. (2009), An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, *in* 'ACM SIGARCH Computer Architecture News', Vol. 37, ACM, pp. 152–163.

Kamil, S., Chan, C., Oliker, L., Shalf, J. & Williams, S. (2010), An auto-tuning framework for parallel multicore stencil computations, *in* 'Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on', IEEE, pp. 1–12.

Ma, W. & Agrawal, G. (2010), An integer programming framework for optimizing shared memory use on GPUs, *in* 'High Performance Computing (HiPC), 2010 International Conference on', IEEE, pp. 1–10.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B. & Hwu, W. W. (2008), Optimization principles and application performance evaluation of a multithreaded GPU using cuda, *in* 'Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming', ACM, pp. 73–82.

Ueng, S.-Z., Lathara, M., Baghsorkhi, S. S. & Hwu, W. H. (2008), CUDA-lite: Reducing GPU programming complexity, *in* 'Languages and Compilers for Parallel Computing', Springer, pp. 1–15.

Zhang, Y. & Mueller, F. (2012), Auto-generation and auto-tuning of 3D stencil codes on GPU clusters, *in* 'Proceedings of the Tenth International Symposium on Code Generation and Optimization', ACM, pp. 155–164.