A Study of the Potential of Locality-Aware Thread Scheduling for GPUs

Cedric Nugteren Gert-Jan van den Braak Henk Corporaal

Eindhoven University of Technology, Eindhoven, The Netherlands
c.nugteren@tue.nl, g.j.w.v.d.braak@tue.nl, h.corporaal@tue.nl

Abstract. Programming models such as CUDA and OpenCL allow the programmer to specify the independence of threads, effectively removing ordering constraints. Still, parallel architectures such as the graphics processing unit (GPU) do not exploit the potential of data-locality enabled by this independence. Therefore, programmers are required to manually perform data-locality optimisations such as memory coalescing or loop tiling. This work makes a case for *locality-aware thread scheduling*: re-ordering threads automatically for better locality to improve the programmability of multi-threaded processors. In particular, we analyse the potential of locality-aware thread scheduling for GPUs, considering among others cache performance, memory coalescing and bank locality. This work does not present an implementation of a locality-aware thread scheduler, but rather introduces the concept and identifies the potential. We conclude that non-optimised programs have the potential to achieve good cache and memory utilisation *when using a smarter thread scheduler*. A case-study of a naive matrix multiplication shows for example a 87% performance increase, leading to an IPC of 457 on a 512-core GPU.

1 Introduction

In the past decade, graphics processing units (GPUs) have emerged as a popular platform for non-graphics computations. Through languages such as OpenCL and CUDA, programmers can use these massively parallel architectures (and other *accelerators*) for computational domains such as linear algebra, image processing and molecular science. The increased popularity of such accelerators has made programming, maintainability, and portability issues of major importance. Although accelerator programming models have partially addressed these issues, programmers are still expected to tune their code for aspects such as (in the case of GPUs) memory coalescing, warp size, core count and the on-chip memories.

To counter the imminent memory wall [3], recent GPUs have been equipped with software-managed on-chip memories (scratch-pad) and hardware-managed on-chip memories (cache). In particular for integrated solutions with general-purpose memories (e.g. ARM Mali, AMD Fusion, XBox One) off-chip memory bandwidth is scarce: using the on-chip memories efficiently is required to exploit the GPU's full potential [13]. In fact, many GPU programs are memory bandwidth intensive: for an example set of benchmarks, this is as much as 18 out of

31 [5]. Specific examples of cache optimisations include cache blocking for sparse matrix vector multiplication (5x speed-up) and *loop tiling* for a stencil computation (3x speed-up). Programmers of GPUs are therefore performing memory coalescing to maximise off-chip throughput or tiling to improve data-locality. Furthermore, programmers determine the allocation of threads to threadblocks, affecting scheduling freedom and cache performance.

With programming models such as CUDA and OpenCL, programmers create a large number of independent[1] threads that execute a single piece of program code (a *kernel*). Still, microprocessors such as the GPU do not exploit the potential of spatial and temporal data-locality enabled by this independence. Therefore, we propose locality-aware thread scheduling: changing the schedule of threads, warps and threadblocks with respect to a kernel's memory accesses.

This work does not aim to improve performance for already optimised (e.g. coalesced, tiled) code, but is instead motivated by non-optimised program code and the performance potential of locality-aware thread scheduling. This improves *programmability*, a metric intertwined with: 1) *portability*: the generality of program code when targeting different microprocessors, 2) *productivity*: the time it costs to design and maintain program code, and 3) *performance*: the speed or energy efficiency of a program. Although the focus of this work lies on GPUs, we make a note that the ideas are equally valid for other cache-based processors that are programmable in an SPMD-fashion.

This work demonstrates that locality-aware thread scheduling can significantly improve the programmability of GPUs. The main contributions are:

- **Section 5:** The potential of multi-level locality-aware thread scheduling for GPUs is identified and quantified for several non-optimised benchmarks.
- **Section 6:** Two example kernels are evaluated further, identifying the effects of thread scheduling on among others caches and memory bank locality.

2 Background

This section briefly introduces the GPU architecture and its execution model. Additional background can be found in the CUDA programming guide [10].

We use NVIDIA's Fermi architecture as an example in this paper. The Fermi architecture has up to 16 cores (also known as *streaming multiprocessors* or *compute units*). Each core contains 32 processing elements (or *CUDA cores*) and a 64KB on-chip configurable memory, combining scratchpad and L1 data cache (16/48KB or 48/16KB). All cores share a larger L2 cache (up to 768KB).

The CUDA and OpenCL programming models allow programmers to specify small programs (*kernels*) that are executed multiple times on different data. Each instance of a kernel (a *thread* in CUDA terminology, a *workitem* in OpenCL terminology) has its own unique identifier. Programmers furthermore divide all their threads in fixed-size blocks (*threadblocks* in CUDA terminology, *workgroups*

[1] Independent apart from explicit per-threadblock synchronisation barriers.

in OpenCL terminology). Threads within a block share an on-chip local memory and can synchronise. However, synchronisation is not possible among blocks.

In a Fermi GPU, a threadblock is mapped in its entirety onto a core. Together, threads from one or more threadblocks can form a set of *active* threads on a single core. For Fermi GPUs, this is limited to 8 threadblocks or 1536 threads, whichever limit is reached first [10]. Such a set of active threads executes concurrently in a multi-threaded fashion as *warps* (NVIDIA) or *wavefronts* (AMD). In Fermi, a warp is a group of 32 threads executing in an SIMD-like fashion on a single core, dividing the workload over processing elements [10].

3 Related Work

Locality-aware thread scheduling has been investigated for non-GPU microprocessors in earlier work. For instance, Philbin et al. [11] formalise the problem of locality-aware thread scheduling for a single-core processor. In other work by Tam et al. [14], threads are grouped based on data-locality for multi-threaded multi-core processors, introducing a metric of *thread similarity*. Furthermore, Ding and Zhong [2] propose a model to estimate locality based on reuse distances. These approaches cannot be applied directly to GPUs, as they do not take into account aspects such as: scalability to many threads, cache sizes, the thread-warp-block hierarchy, nor the active thread count.

Recent work on GPUs has investigated the potential of scheduling less active threads to improve cache behaviour. Kayiran et al. [5] propose a compute/memory-intensity heuristic to select the active thread count. Furthermore, Rogers et al. [12] propose a hardware approach: the number of active threads is adapted at run-time based on *lost locality* counters. However, these works only consider active thread count reduction: they do not investigate thread scheduling.

Current scheduling research for GPUs is in the context of divergent control flow rather than data-locality. By dynamically regrouping threads into warps, those following the same execution path can be scheduled together. Dynamic warp formation in the context of memory access coalescing is discussed in e.g. [6, 7]. Recent work has focussed on two-level warp scheduling to reduce the impact of memory latency [4, 8]. Although we not address control flow, we note that an ideal scheduler takes both aspects (data-locality and control flow) into account.

4 Experimental Setup

The experiments in this work are performed with GPGPU-Sim 3.2.1 [1] using a GeForce GTX580 configuration (Fermi) with a 16KB L1 cache (128 byte cache-lines) and a 768KB L2 cache. The GTX580 has 16 SIMT cores (or SMs) for a total of 512 *CUDA cores*. From the simulation results we report IPC (higher is better)[2], cache miss rates (lower is better), and load balancing amongst off-chip memory banks (higher is better).

[2] IPC (instructions per cycle) is counted as the throughput of scalar operations and load/store instructions over all CUDA cores and load/store units in the GPU.

4.1 Implementation in GPGPU-Sim

The GPGPU-Sim simulator was modified to perform the thread scheduling experiments presented in this work. The run-time scheduling mechanism of a GPU (and of the simulator) is non-trivial, including multiple hierarchies and dynamic aspects (e.g. influenced by memory latencies). This mechanism is therefore kept intact in GPGPU-Sim. Instead, this work implements a pre-processing 'mapping' step to the thread and block identifiers. This mapping step takes thread identifiers t_i and block identifiers b_i and calculates new identifiers as $t'_i = f(t_i, b_i)$ and $b'_i = g(t_i, b_i)$. The functions $f()$ and $g()$ implement alternative thread schedules as will be further discussed in Sect. 5.1. Because the mapping is applied before the hardware run-time thread scheduling, the effect is equivalent to applying the $f()$ and $g()$ to the software thread and block identifiers - a task currently assigned to CUDA and OpenCL programmers.

4.2 Benchmark Selection

This paper includes results for 6 non-optimised CUDA benchmarks, i.e. sub-optimal implementations rather than fine-tuned benchmarks (e.g. Parboil or Rodinia). The main reason for this choice is that this work aims to improve the programmability of the GPU rather than the maximum performance. In other words, if performance of these naive non-optimised benchmarks can be improved without having to change the program code, GPU acceleration is made available to a wider audience ('non-ninja programmers'). Even expert programmers can benefit from increased flexibility and require fewer optimisations to achieve the full potential of the GPU.

The benchmarks are: the computation of an integral image (row-major and column-major), a 2D convolution (11 by 11), a 2D matrix copy (each thread copies either a row or a column), and a naive matrix-multiplication. Image and matrix sizes are 512 by 512. Fig. 1 illustrates their memory access patterns:

1. **Integral image (row-wise):** Every thread at coordinates (x, y) in a 2D image produces a single output pixel at (x, y) by consuming all input pixels (x', y) for which $x' \leq x$. In the example, thread 0 consumes input 0 (red), thread 1 consumes inputs 0 and 1 (red and blue), and so on.
2. **Integral image (column-wise):** Equal to the row-wise version, but each thread instead consumes all input pixels (x, y') for which $y' \leq y$.
3. **11 by 11 convolution:** Each thread produces a single pixel in a 2D image by consuming an input pixel at the same coordinates (blue) and its neighbourhood of $(11 \cdot 11) - 1$ elements (green).
4. **Matrix-multiplication:** Each thread with coordinates (x, y) consumes a row $(*, y)$ of an input matrix and a column $(x, *)$ of another input matrix to produce a single element in an output matrix at (x, y).
5. **Matrix copy (per row):** Each thread consumes a row of an input matrix to produce the corresponding row in an output matrix.
6. **Matrix copy (per column):** As before, but now columns instead of rows.

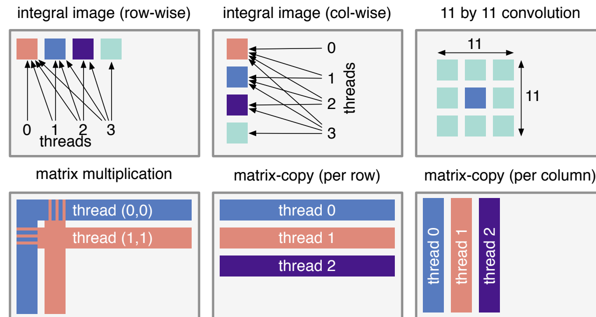

Fig. 1. Illustrating the memory access patterns of the 6 benchmarks.

5 Quantifying the Potential

Many GPU programs contain a large number of independent threads that can be freely re-ordered. This re-ordering (changing the thread schedule) is motivated by the following data-locality performance optimisations: 1) multiple threads accessing a single cache-line must be grouped in a warp (memory coalescing), 2) threads having strong inter-thread locality must be grouped within a single threadblock (sharing a L1 cache), 3) threadblocks with data-locality must be executed either on a single core in temporal vicinity or simultaneously on different cores (sharing a L2 cache), 4) threads executing simultaneously must minimise pollution of the shared caches, and 5) threads executing simultaneously must spread their accesses as evenly as possible across the memory banks.

Consider an SPMD (single-program multiple-data) kernel with n threads $t_1, t_2, ..., t_n$, each referencing a number of data elements. This work assumes that all n threads are independent[3] and can be reordered as $r = n!$ distinct sequences $s_1, s_2, ..., s_r$. The problem of locality-aware thread scheduling is to find a sequence s_i of n threads such that execution time is minimal. On a GPU, thread scheduling influences execution time in terms of efficient use of the caches, memory coalescing, memory bank locality, and the number of active threads.

5.1 Candidate Thread Schedules

Various thread schedules are tested in GPGPU-Sim to quantify the potential of locality-aware thread scheduling. Because the number of threads n is typically large (e.g. 2^{20}), it is impractical to test all r orderings. Therefore, only a limited set of schedules is considered: schedules with regularity and structure, matching the target regular and structured programs. The selected schedules are illustrated in Fig. 2 and briefly discussed below. Note that these schedules represent the mapping step discussed in Sect. 4.1 and are still subject to the GPU's multi-level scheduling mechanism. The schedules are:

[3] Dependences (e.g. barriers) can be added as constraints on the thread ordering.

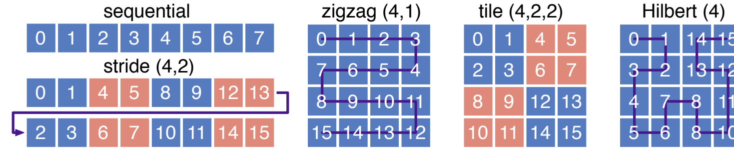

Fig. 2. Examples with 8 or 16 threads. The numbering shows the new sequence and the layout the original sequence (left-to-right, top-to-bottom).

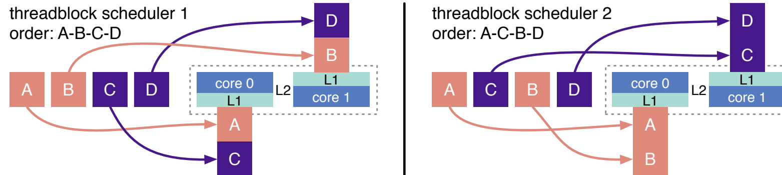

Fig. 3. Two schedulers for threadblocks A–D, assuming locality between A and B (red) and between C and D (purple). The results are L2 locality (left) or L1 locality (right). The arrows represent the GPU scheduler applied after our 'mapping' (or ordering).

1. **Sequential:** The unmodified original ordering, i.e. $f(x) = x$ and $g(x) = x$. Note that, although it is a sequential ordering from a pre-processing perspective, the actual ordering is still subject to the GPU's thread, warp, and block scheduling policies.
2. **Stride(a, b):** An ordering with a configurable stride (a) and granularity (b) (e.g. warp or threadblock granularity) with respect to the original ordering. Strided schedules have the potential to e.g. ameliorate bad choices of a 2D-coordinate to thread mapping [13].
3. **Zigzag(a, b):** An ordering assuming a 2D grid of threads, reversing the ordering of odd rows. The parameters are the row-length (a) and the granularity (b). Zigzag can exploit 2D locality, but might degrade coalescing for small granularities.
4. **Tile(a, b, c):** 2D tiling in a 2D grid. Tiling takes as parameter the length of a row (a) and the dimensions of the tile (b x c). It has been shown that tiling has potential to exploit locality on GPUs [13].
5. **Hilbert(a):** A space filling fractal for grids of size a by a with 2D locality.

Two threadblock-schedulers are implemented on top of the candidate schedules (Fig. 3): either schedule threadblocks over cores in a round-robin fashion (left) or allocate subsequent threadblocks to subsequent cores (right). In case threadblocks with locality are grouped close to each other, the first threadblock-scheduler can benefit from locality in the L2 cache (in space among cores), while the second can benefit from locality in L1 (in time among threadblocks).

Our experiments consider a subset of 2170 schedules. This includes a sweep over the 5 orderings, several small power-of-2 parameter values (e.g. stride-size), the two threadblock-schedulers, and 5 active thread counts (64, 128, 256, 512, 1024) to identify the trade-offs between cache contention and parallelism [5, 12].

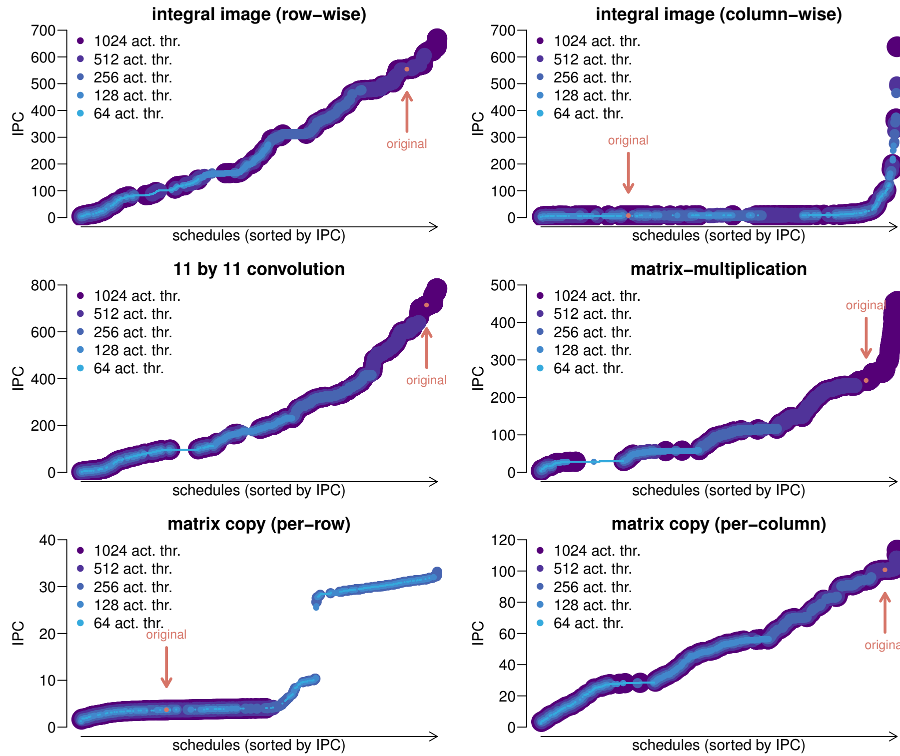

Fig. 4. Sorted IPC results (higher is better) from GPGPU-Sim for 2170 schedules per benchmark. The vertical red arrow identifies the original schedule (no changes applied to GPGPU-Sim). Darker and larger glyphs represent more active threads, lighter and smaller glyphs represent fewer active threads.

5.2 Experimental Results

Fig. 4 gives the IPC results when simulating all candidate schedules for the benchmarks with GPGPU-Sim. Each set of 2170 results is sorted by their achieved IPC. The original (unmodified) schedule is highlighted, its horizontal position indicating the performance potential for a particular benchmark. Note that these graphs are meant to identify the main shape of the '*landscape*', detailed results are presented in Sect. 6. We observe the following:

1. **Integral image (row-wise):** There is a wide performance variation among the different schedules: IPC ranges from 2 to 700. The default schedule is already performing well: it has coalesced memory accesses and uses the caches efficiently. Still, there is opportunity for a 20% performance improvement, achieved for example by using a 8 by 16 tiled schedule. The active thread count is not strongly correlated to performance. Even so, the best 5% schedules all use 1024 active threads.

2. **Integral image (column-wise):** The default schedule at an IPC of 7 is suffering from uncoalesced memory accesses and bad cache locality for this purposely poorly design kernel. Using a schedule with a stride equal to the width of the image resolves these problems, bringing performance back to the level of the row-wise integral image computation.
3. **11 by 11 convolution:** The overall results look similar to the row-wise integral image case at first glance. However, inspection of the results shows that the best candidates are zigzag as opposed to tiled schedules, achieving up to 10% improvement over the default.
4. **Matrix-multiplication:** The results show that there is up to 87% to gain over the default schedule in terms of performance (see Sect. 6.1 for details).
5. **Matrix copy (per row):** The active thread count is of significant importance, although the performance is in general low due to the cache and memory unfriendly assignment of work to threads. Schedules with 512 or 1024 active threads (including the default) yield an IPC of 5 at best, while schedules with 64, 128, or 256 active threads achieve an IPC of up to 34. This is the only test-case where more threads does not yield better performance.
6. **Matrix copy (per column):** Better overall performance compared to per-row copy. Sect. 6.2 analyses the results and the 12% potential in detail.

Note that in contrast to the two integral image cases, it is not possible to achieve equal performance for the two matrix copy cases. The reason is the integral image's flexibility: each thread computes a single result. In contrast, matrix copy processes (in our implementation) an entire row / column per thread, limiting the scheduling freedom: we do not consider changing the workload *within* a thread.

The same testing methodology was applied to several other naive benchmarks. An example is the computation of an 8 by 8 discrete cosine transform (DCT) on a 2048 by 2048 input using a nested for-loop in the kernel body with 64 iterations. A sweep through the different thread schedules led to a 3.2x speedup (an increase from an IPC of 175 to 570) using a schedule with a stride of 512 at a granularity of 8, moving multiple groups of threads belonging to one 8 by 8 transform (64 threads) together into a single threadblock.

Similarly, a symmetric rank-k kernel from PolyBench shows a 3 times speedup. Several other tested benchmarks have not shown significant changes at all. This includes matrix-vector summation from the PolyBench benchmark and the breath-first-search and SRAD kernels from Rodinia. These results were expected, as closer inspection of these benchmarks shows already optimised code.

6 Two Case Studies

Sect. 5 illustrated that the performance potential varies from limited (e.g. 10% for the convolution benchmark) to significant (e.g. 87% for matrix-multiplication). We also saw different best schedules for different benchmarks and a varying correlation between performance and active thread count. To get additional insight, this section discusses two of the benchmarks in more detail. We only present a subset of the data due to the large quantity (schedules, benchmarks, metrics).

Fig. 5. Simulation results for the matrix-multiplication example for strided schedules. Shown are the IPC (higher is better) and the L1 and L2 miss rates (lower is better).

6.1 Matrix-Multiplication

Matrix-multiplication is one of the examples that shows a significant performance potential (up to 87%) from its default IPC of 245. To identify the reason why certain schedules perform better than others, we take a detailed look at the simulation results for the strided schedules. Because the stride ordering has two parameters (`P1` for the stride and `P2` for the granularity), the data can be visualised as a 2D heatmap. Fig. 5 shows the heatmaps for the IPC and the L1 and L2 cache miss rates, as well as their correlation.

Fig. 5 shows a high inverse correlation (-0.8) between the IPC and the L1 miss rate: the 4 best candidates (with IPC > 300) all have the lowest L1 miss rate (16%). Although a low L2 miss rate also contributes to a high IPC, Fig. 5 (bottom right) shows a lower correlation. The results of Fig. 5 can be explained after detailed investigation. First of all, schedules with a small granularity (`P2` < 32) can reduce the amount of coalescing significantly, leading to a low IPC and high cache miss rates. Second, schedules with a large stride and a large granularity form small 'tiles' in the 2D space of the matrix, improving locality. Finding the best tile dimensions is non-trivial and dependent on among others matrix dimensions and cache configuration. In this case, a ratio of 8:1 for `P1` and `P2` yields the best results for L1 and 2:1 for the L2 cache.

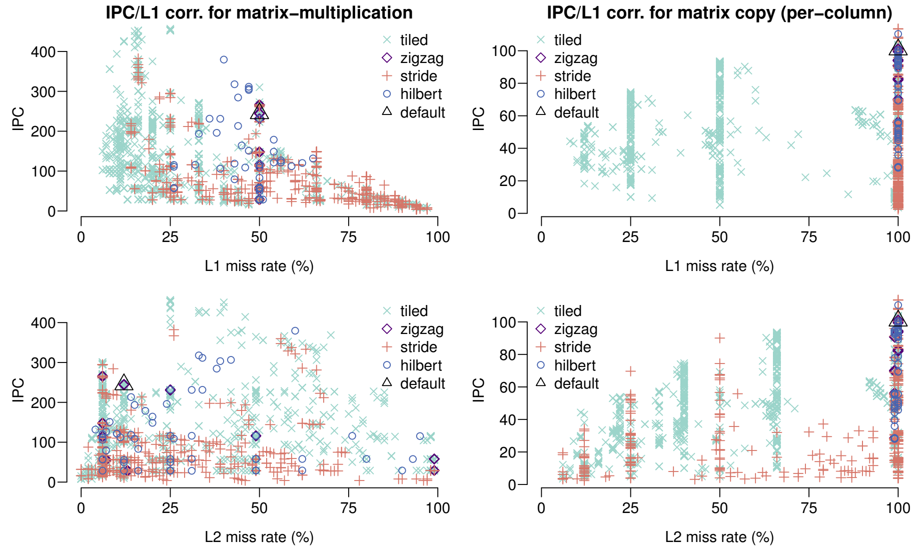

Fig. 6. Correlation plots for IPC (higher is better) and cache miss rates (lower is better) for the matrix-multiplication example (left) and the per-column matrix copy (right). Different colours/shapes represent different schedule types.

The left hand side of Fig. 6 shows the correlation plots of all the 2170 schedules for the matrix-multiplication example for 3 metrics: the top graph shows the correlation between IPC (y-axis) and L1 miss rate (x-axis), the bottom between IPC and L2 miss rate. From these results, we observe that the strided and tiled schedules have similar behaviour: they both cover the entire IPC and miss rate spectrum and show a high correlation between the IPC and L1 miss rate. We also observe a large amount of schedules with a L1 cache miss rate of around 50%, including the default and zigzag schedules. The best result uses 32x2 tiles with a width of 2048 and the first scheduler.

6.2 Per-Column Matrix Copy

The correlation plots for the per-column matrix copy are shown on the right hand side of Fig. 6. From these plots, we observe that the IPC and cache miss rates are not as correlated as in the matrix-multiplication example. In fact, the best performing schedules have L1 and L2 cache miss rates of 100%. We furthermore observe that L1 cache miss rates only vary for tiled schedules and that most of them are distributed in a log_2 fashion: they have values of 100%, 50%, 25% and 12.5%. These 'improved' miss rates are cases where a lowered ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$) memory coalescing rate results in additional cache hits.

Unlike the matrix-multiplication example, cache miss rates are not correlated with the IPC. Therefore, Fig. 7 focuses on other aspects: it shows the IPC and DRAM bank efficiency (the fraction of useful over theoretical accesses) for

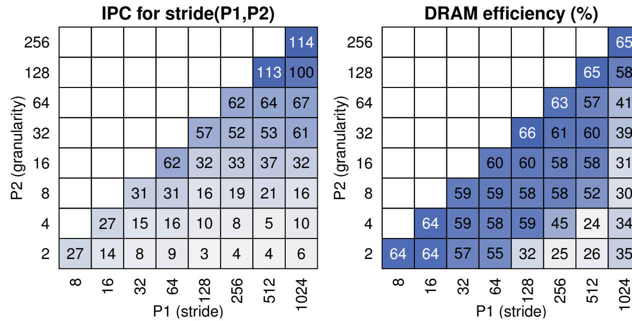

Fig. 7. Simulation results for the per-column matrix copy using strided schedules. Shown are IPC and DRAM bank efficiency (higher is better).

strided schedules. A low DRAM bank efficiency can be the cause of an uneven distribution of accesses over the DRAM banks (6 for the simulated GPU): certain phases of the benchmark access only a subset of DRAM banks, limiting the DRAM throughput. Although DRAM bank efficiency is correlated to the IPC, Fig. 7 also shows that it is not the only contributing effect. As with matrix-multiplication, memory access coalescing[4] also plays a role, explaining the low IPC for `P2` < 32. DRAM efficiency can still be high in this case, as the number of accesses is increased as well.

7 Summary and Future Work

This work identified the potential for locality-aware thread scheduling on GPUs: re-ordering threads to increase data-locality and subsequently performance and energy efficiency. 2170 candidate schedules were simulated for 6 non-optimised CUDA benchmarks, showing a performance potential varying from 10% to multiple orders of magnitude. The benchmarks were explicitly chosen to be non-optimised: enabling competitive performance for such benchmarks will greatly improve the programmability. Our study has also identified aspects to consider: cache miss rates, coalescing, bank locality, and the number of active threads. An example is a straightforward implementation of matrix multiplication, which achieved a 87% performance increase by modifying the thread schedule.

Although this work has shown that locality-aware thread scheduling has the potential to improve programmability (better performance without changing the code), we have also shown that it is non-trivial to find the best thread schedule. This work can therefore be seen as a first step towards an investigation of how to find a good thread schedule: the ideas are presented, the potential has been shown, but an implementation has not been presented. A solution could potentially be found by evaluating schedules using a complete or partial performance model. This is motived by the detailed studies in this work, which have shown

[4] Coalescing is not visualised because GPGPU-Sim lacks the corresponding counters.

that performance is correlated to one or more metrics such as memory access coalescing or cache miss rate. An example of this is the use of the L1 cache model presented in [9]. Another possibility would be to iterate efficiently through all schedules, for example through auto-tuning (evaluating specific schedules on actual hardware), machine learning / neural networks (pre-training a model), or using hardware counters to dynamically change schedules (e.g. [12]).

References

1. A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads using a Detailed GPU Simulator. In *ISPASS: International Symposium on Performance Analysis of Systems and Software*. IEEE, 2009.
2. C. Ding and Y. Zhong. Predicting Whole-Program Locality through Reuse Distance Analysis. In *PLDI-24: Conference on Programming Language Design and Implementation*. ACM, 2003.
3. S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *IEEE Computer*, 44, 2011.
4. M. Gebhart, D. Johnson, D. Tarjan, S. Keckler, W. Dally, E. Lindholm, and K. Skadron. A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors. *ACM Trans. on Computer Systems*, 30:8:1–8:38, 2012.
5. O. Kayiran, A. Jog, M. Kandemir, and C. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT-22: International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2013.
6. A. Lashgar, A. Baniasadi, and A. Khonsari. Dynamic Warp Resizing: Analysis and Benefits in High-Performance SIMT. In *ICCD-30: International Conference on Computer Design*. IEEE, 2012.
7. J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *ISCA-37: International Symposium on Computer Architecture*. ACM, 2010.
8. V. Narasiman, M. Shebanow, C. Lee, R. Miftakhutdinov, O. Mutlu, and Y. Patt. Improving GPU Performance via Large Warps and Two-level Warp Scheduling. In *MICRO-44: International Symposium on Microarchitecture*. ACM, 2011.
9. C. Nugteren, G.-J. v. d. Braak, H. Corporaal, and H. Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *HPCA-20: International Symposium on High Performance Computer Architecture*. IEEE, 2014.
10. NVIDIA. *CUDA C Programming Guide 5.5*, 2013.
11. J. Philbin, J. Edler, O. Anshus, C. Douglas, and K. Li. Thread Scheduling for Cache Locality. In *ASPLOS-7: International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1996.
12. T. Rogers, M. O'Connor, and T. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO-45: International Symposium on Microarchitecture*. IEEE, 2012.
13. J. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W. Hwu. Optimization and Architecture Effects on GPU Computing Workload Performance. In *INPAR: Workshop on Innovative Parallel Computing*. IEEE, 2012.
14. D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys-2: European Conference on Computer Systems*. ACM, 2007.