Predictable Thread Coarsening

NICOLAI STAWINOGA and TONY FIELD, Imperial College London

Thread coarsening on GPUs combines the work of several threads into one. We show how thread coarsening can be implemented as a fully automated compile-time optimisation that estimates the optimal coarsening factor based on a low-cost, approximate static analysis of cache line re-use and an occupancy prediction model. We evaluate two coarsening strategies on three different NVidia GPU architectures. For NVidia reduction kernels we achieve a maximum speedup of 5.08x, and for the Rodinia benchmarks we achieve a mean speedup of 1.30x over 8 of 19 kernels that were determined safe to coarsen.

CCS Concepts: • **Computer systems organization** → **Parallel architectures**; • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: GPU, compiler optimisations

ACM Reference format:

Nicolai Stawinoga and Tony Field. 2018. Predictable Thread Coarsening. *ACM Trans. Archit. Code Optim.* 15, 2, Article 23 (June 2018), 26 pages.
<https://doi.org/10.1145/3194242>

1 INTRODUCTION

GPU architectures are an indispensable provider of parallel processing power, offering large-scale data parallelism and floating point performance that can far exceed that of conventional architectures. To realise the potential performance offered by GPUs, much research has been undertaken into compiler optimisations specifically tailored to them. One such optimisation is *thread coarsening*, in which code that is normally executed by several different threads is merged into a single thread; the effect is to execute a smaller number of larger, i.e., more coarse-grained, threads than before.

Thread coarsening affects a reduction in parallelism in the application, which can have both a beneficial and a detrimental effect on runtime requirements. To illustrate this, Figure 1 shows an example kernel before and after applying the transformation with a coarsening factor of two. All instructions that are dependent on the thread ID (`idx`) are duplicated, whilst all other instructions are shared between the two coarsened instances. One such “shared instruction” is a synchronisation barrier (line 8).

The transformed code in Figure 1(b) outperforms the original code for two reasons:

- (1) As the coarsened version of the code requires only half the number of threads to be launched, the barrier is executed half as many times, when taken across the whole program execution.

Authors’ addresses: N. Stawinoga and T. Field, Department of Computing, Huxley Building, 180 Queen’s Gate, Imperial College London, London SW7 2AZ, United Kingdom; emails: {n.stawinoga13, t.field}@imperial.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 ACM 1544-3566/2018/06-ART23 \$15.00

<https://doi.org/10.1145/3194242>

<pre> 1 kernel foo(global float* in, out) { 2 uint idx = get_global_id(0); 3 4 float data = read_data(in, idx); 5 6 float result = do_some_work(data); 7 8 barrier(); 9 write_data(out, idx, result); 10 11 }</pre>	<pre> 1 kernel foo(global float* in, out) { 2 uint idx0 = 2 * get_global_id(0); 3 uint idx1 = 2 * get_global_id(0) + 1; 4 float data0 = read_data(in, idx0); 5 float data1 = read_data(in, idx1); 6 float result0 = do_some_work(data0); 7 float result1 = do_some_work(data1); 8 barrier(); 9 write_data(out, idx0, result0); 10 write_data(out, idx1, result1); 11 }</pre>
(a) Original code	(b) Coarsened code

Fig. 1. Applying thread coarsening by a factor of two.

- (2) The increased number of instructions that will be generated from the coarsened code increases the scope for exploiting hardware instruction-level parallelism.

These effects scale to some extent with increasing coarsening factor, although there will come a point where the reduction in the number of threads limits the ability of the application to exploit the available parallelism efficiently. Increasing the coarsening factor also raises a kernel's resource consumption, e.g., of registers, eventually resulting in reduced occupancy. An increased workload per thread can also increase the pressure on the cache in some kernels (see Section 4). Thus, there is a performance tradeoff, and the challenge is both to determine whether coarsening should be applied to a given kernel, and if so, what the optimal coarsening factor is, i.e., the optimal number of threads to merge that maximises the overall performance.

Thread coarsening as an optimisation has been extensively explored as a manual, semi-automatic, auto-tuning-based, and machine-learning-based optimisation (cf. Magni et al. (2013, 2014a), Unkule et al. (2012), and Volkov and Demmel (2008), respectively). It has also been reviewed by Stratton et al. (2012) as one of eight GPU-specific code optimisations, none of which—to the best of our knowledge—is currently implemented in a commercially available general-purpose GPU compiler. The goal of this article is to outline what key obstacles remain that currently prevent thread coarsening from being used as a fully automated compile-time optimisation, and provide solutions to overcome these obstacles. The intention is that this will allow thread coarsening to be integrated in commercially available general-purpose compilers.

In this article, we propose a static compile-time method for estimating the optimal coarsening factor of a kernel based on a simple occupancy metric. The key point is that we avoid the need to execute different coarsened variants, as would be required by runtime methods such as auto-tuning. Our occupancy-based model combines architectural knowledge of various different NVidia GPU architectures with knowledge of the code.

In addition, we propose a way of filtering out kernels on which thread coarsening may have a potentially detrimental effect due to increased cache pressure. This filtering could in theory be solved by re-use distance analysis (Almásí et al. 2002), but this is expensive in terms of both runtime and memory requirements (Nugteren et al. 2014). We show that by focusing only on cache line re-use, rather than re-use distance, these limitations can be overcome, making the analysis efficient enough to become tractable for an optimising compiler. To the best of our knowledge, this heuristic approach is the first of its kind to be implemented for GPUs.

Our contributions are as follows:

- We contrast two different modes of thread coarsening that increase granularity within a single block of threads (*thread-level coarsening*) and across multiple blocks (*block-level coarsening*) (Section 3).

- We introduce a static heuristic analysis to identify and filter out kernels where thread coarsening is likely to have a negative effect on performance because of cache pressure (Section 4).
- We present an occupancy-based performance model for estimating optimal coarsening factors based on the above static analysis and re-compilation, i.e., without requiring execution of the kernel body (Section 5).
- We evaluate our approach using benchmarks from the Rodinia benchmark suite as well as seven reduction kernels from the NVidia OpenCL SDK that have been manually optimised to different degrees (Section 6). The evaluation shows that speedups of up to 5.08x (average 1.30x) can be achieved for kernels that are deemed safe to coarsen.

We implement our work as an extension to the existing LLVM compiler-based tool chain presented in Magni et al. (2013). Our software is publicly available on GitHub.¹

2 GPU ARCHITECTURES AND PROGRAMMING

This section gives a brief overview of architectural concepts employed in NVidia GPUs that are relevant to the article, both from a hardware and software perspective. Exact architectural specifications can be found in Table 2 below.

2.1 Streaming Multiprocessors

An NVidia GPU holds several *Streaming Multiprocessors* (SMs), each of which forms one independent OpenCL *compute unit*. An SM is responsible for executing thread blocks (work-groups, in OpenCL terminology). For brevity, we will sometimes refer to a thread block as simply a block.

Several thread blocks may be executed on one SM simultaneously, but a block cannot be split over several SMs. As such, SMs hold all hardware required for the processing of thread blocks, including single- and double-precision processing cores, instruction buffers, *warp schedulers* responsible for scheduling threads, SIMD style, in chunks of 32 (called a *warp*), a register file, on-chip memory, and so on. Since the introduction of the Kepler architecture, each SM can hold up to 2,048 threads simultaneously. The number of blocks concurrently executable on one SM is limited and the limit is specified in the CUDA/OpenCL Compute Capability of the respective device.

The architecture of one SM is thus similar to an SIMD (Single Instruction Multiple Data) processor, with a warp being the vector length. However, because of the added multi-threading capabilities, it is often referred to as SIMT (Single Instruction Multiple Threads).

2.2 Memory Hierarchy

The memory subsystem of a GPU consists of a large DRAM base acting as the *global memory*, several layers of caches, as well as registers. The global memory acts as the device's main memory. It is mounted on-board the GPU, holds data that can be accessed by all SMs, and is typically several GB large.

The L2 cache services all requests that target the global memory, acting as a point of data unification between SMs. The architectures employed in our experiments have a cache size of 1.5–4MB. Given the large number of threads that this cache potentially services, this cache can fill up quickly in some applications.

Each SM unit holds its own L1 cache as well as *shared memory* (*local memory* in OpenCL terminology), which is a software-managed cache. In earlier architectures, up to and including Kepler, the L1 cache and shared memory used to share the same physical 64KB memory, leaving it up

¹See <http://github.com/n-io/autocoarsening>.

to the application to decide whether to have more of the one or of the other. Beginning with the Maxwell architecture, the cache hierarchy was changed and the two were separated, with SMs having received a dedicated pool of shared memory (cf. NVIDIA Corporation (2016)).

Each SM has a register file of a fixed number of registers, with limits in place for the number of registers that can be used per thread and per block. Registers are assigned statically to blocks in chunks equal to the warp size. Because this assignment is static, register spilling is decided at compile time. By default, register spilling is disabled on NVidia GPUs, which is the configuration we use throughout our experiments. For resource-intensive kernels, the hardware will instead schedule fewer thread blocks per SM for concurrent execution (i.e., reduce the *occupancy*, see below).

As threads belonging to the same warp are executed in lock-step, the hardware seeks to combine memory accesses to the same cache line. This process is called *memory coalescing*. If all bytes of a cache line are referred to, the access is *fully coalesced*.

2.3 Thread Block Scheduling

The block scheduler assigns blocks of threads to SMs to be executed. Incoming blocks are taken from a processing queue, from which they may be scheduled in any order (cf. NVidia (2010)). The implication is that no assumptions should be made inside a kernel that would require a specific order in the scheduling of blocks.

The block scheduler seeks to schedule the maximum number of thread blocks for concurrent execution, that is, to maximise the *occupancy*, although this is not always guaranteed to maximise performance (see Volkov (2010)).

2.4 Occupancy

Occupancy is an alternative measure to hardware utilisation of a specific kernel, concerned only with the utilisation of active threads across the device. As such, occupancy is defined by the number of active threads as a proportion of the maximum number of active threads that could be executed on one SM. For example, if 1,536 active threads are scheduled onto an SM that can handle a maximum of 2,048 active threads, the occupancy is 0.75 or 75%. A distinction exists between *theoretical* and *achieved* occupancy. The *theoretical occupancy* captures the upper bound of occupancy that a compiled kernel can achieve on an SM. This can be statically determined by matching a kernel's resource requirements against hardware specifications, in particular register usage, shared memory usage, and requested thread block size. The *achieved occupancy* of a kernel depends on additional runtime factors that may cause parts of the hardware to idle, such that the occupancy observed is lower than its theoretical upper bound. This would occur in the presence of unbalanced workloads, e.g., where a block or a warp within a thread block finishes earlier than another (referred to as a *tail effect*), or toward the end of a computation where some SMs will have no work to do. The achieved occupancy thus depends on the problem size, e.g., for small problem sizes there may be insufficient work to enable the machine to reach 100% occupancy.

Unless otherwise specified, we use the term occupancy to refer exclusively to theoretical occupancy. However, where indicated, we also take the problem size into consideration, in which case we refer to the achievable (rather than achieved) occupancy.

2.5 GPU Programming

The programming model specified by the OpenCL API (and the related CUDA language) has become the de facto standard for handling the large-scale parallelism offered by GPUs and other parallel architectures. An OpenCL kernel is a method runnable on a GPU. It describes the processing of one parallelisable *work-item* belonging to one *work-group*. Work-items and work-groups

OpenCL function	Description
<code>get_global_id</code>	Global work item ID value
<code>get_global_size</code>	Number of global work items
<code>get_local_id</code>	Local work item ID
<code>get_local_size</code>	Number of local work items
<code>get_group_id</code>	Work group ID
<code>get_num_groups</code>	Number of work groups
<code>get_work_dim</code>	Number of dimensions in use

(a)

```

1  for groupIdZ = 0 .. get_num_groups(2)
2  for groupIdY = 0 .. get_num_groups(1)
3  for groupIdX = 0 .. get_num_groups(0)
4  for localIdZ = 0 .. get_local_size(2)
5  for localIdY = 0 .. get_local_size(1)
6  for localIdX = 0 .. get_local_size(0)
7  __kernel void foo() { ... };

```

(b)

Fig. 2. (a) OpenCL work-item built-in functions. (b) Implied loop nest within which kernels are executed.

are abstract software concepts that, for an NVidia platform, are mapped directly onto the hardware's threads and thread blocks, respectively. Synchronisation may only happen within work-groups, although wider-reaching synchronisation can be achieved by splitting a kernel in several parts, which are separately launched. Work-items belonging to the same local work-group can also jointly use *local memory* to enable data-sharing between threads; this corresponds to *shared memory* in NVidia's terminology.

The *launch configuration* of a kernel, which must be specified when a kernel is placed on the processing queue, specifies the number of global and local work-items. Both are number triplets to represent up to three dimensions. The launch configuration thus specifies an (up to) three-dimensional processing grid (global size), within which the kernel will be executed in chunks equal to the specified local size. The OpenCL API specifies a number of work-item related built-in functions (see Figure 2(a)) that allow each thread to identify the "location" of its associated work-item within the processing grid, as well as to provide information on the dimensions of the grid.

Kernels may be thought of as executing in an implied loop nest, as illustrated in Figure 2(b). The coarsening optimisation discussed in this article (cf. Section 3) effectively means unrolling one of the loops in the implied loop nest. The choice of loop to be unrolled affects the dimension and mode of coarsening. The present article focuses on NVidia GPU platforms so we will use thread and thread block instead of work-item and work-group, respectively, although it is important to understand that when executing OpenCL kernels on NVidia GPUs they refer to the same concept.

3 THREAD COARSENING

The evaluation we present in Section 6 compares two coarsening strategies: *Thread-level coarsening* applies coarsening on the level of individual threads, combining two or more threads from the same block, whilst *block-level coarsening* combines the work of several thread blocks into one. Thread-level coarsening was the focus of Magni et al. (2013). The idea of block-level coarsening was mentioned in Unkule et al. (2012) but without details of the required code generation rules or performance evaluation.

3.1 Thread-Level Coarsening

In thread-level coarsening, each thread block performs the same amount of work but with fewer threads. The total number of thread blocks therefore remains unchanged, while the number of threads per block and the number of threads in total are reduced. This does not necessarily result in an immediate reduction in occupancy as more, smaller thread blocks can be scheduled simultaneously if sufficient resources are present. However, as each SM has limitations in terms of registers, shared memory, and concurrently runnable thread blocks, a reduction in occupancy is bound to occur eventually.

Table 1. Implementation Details for Thread and Block-Level Coarsening, with *Stride S*, Coarsening Factor *C*, and *i* as the Index of the Coarsened Thread or Thread Block Such That $0 \leq i < C$

OpenCL function	After thread-level coarsening	After block-level coarsening
get_local_id()	$\lfloor \frac{\text{get_local_id}()}{S} \rfloor * S * C$ + get_local_id() * mod(S) + i * S	get_local_id()
get_group_id()	get_group_id()	$C * \text{get_group_id}() + i * S$
get_global_size()	$C * \text{get_global_size}()$	$C * \text{get_global_size}()$
get_local_size()	$C * \text{get_local_size}()$	get_local_size()
get_num_groups()	get_num_groups()	$C * \text{get_num_groups}()$
get_global_id()	(rewrite using def: get_group_id() * get_local_size() + get_local_id())	

To combine the workload of several threads into one, it is necessary to specify which threads are to be merged. This is controlled via the *stride* parameter, acting as an offset between the IDs of threads that are to be combined.

To ensure that duplicated thread IDs calculated by using the stride value are within the boundaries of the thread block, the stride *S* must be constrained by

$$S \leq \frac{\text{get_local_size}(d)}{C}, \quad (1)$$

where *d* is the dimension in which coarsening is applied, and *C* is the coarsening factor. When choosing the stride parameter it is important to avoid breaking any existing memory coalescing (Magni et al. 2013). On NVidia architectures, *S* should be at least as large as the warp size, to ensure that any memory coalescing (Section 2.2) exploited by an uncoarsened kernel is retained in its coarsened equivalent.

In the kernel body, code duplication is based on calls to *get_local_id()*, and includes all dependent instructions. Rewrite rules for relevant OpenCL functions are outlined in Table 1.

Dealing with allocated shared memory resources under thread-level coarsening is straightforward, as shared memory is allocated on a per-block basis in the OpenCL programming model. Although the actual number of threads in a block will change as a result of coarsening, each block still needs to perform the work of all threads specified in the original launch configuration. Consequently, shared memory allocations inside a kernel are left unaltered, as the same amount of shared memory per block is required. However, as discussed, shared memory requirements per SM may increase due to an increase in the number of concurrently executable blocks per SM.

The first two code blocks of Figure 3 show the original version of a reduction kernel (*reduce2*) from the NVidia OpenCL SDK together with the coarsened version (*thread_coarsened*), built using the thread-level coarsening rules of Table 1. The kernel is one-dimensional, as indicated by the zero-argument with which OpenCL built-in functions are invoked. It can be seen that all instructions depending on *get_local_id(0)* have been duplicated, including code regions whose entry condition depends on a duplicated value, such as the conditional inside the for-loop. The chosen value for the stride is 32. Note that for presentational purposes only, the rewrite of the *i* variable has been simplified to avoid repeating the expression of the previous line. Since the kernel will be executed by half the number of threads per block, the value of *get_local_size(0)* has been scaled by a factor of two, to yield the original value. No changes are required to the use of the shared memory resources.


```

1  #define BLOCK_SIZE 512
2  #define T double
3
4  __kernel void reduce2(__global T *g_idata, __global T *g_odata, int n) {
5      unsigned int tid = get_local_id(0);
6      unsigned int i = get_global_id(0);
7      __local T sdata[BLOCK_SIZE];
8      sdata[tid] = (i < n) ? g_idata[i] : 0;
9      barrier(CLK_LOCAL_MEM_FENCE);
10
11      // do reduction in shared mem
12      for(unsigned int s=get_local_size(0)/2; s>0; s>=1) {
13          if (tid < s) {
14              sdata[tid] += sdata[tid + s];
15          }
16          barrier(CLK_LOCAL_MEM_FENCE);
17      }
18
19      // write result for this block to global mem
20      if (tid == 0) g_odata[get_group_id(0)] = sdata[0];
21  }

```

```

1  __kernel void thread_coarsened(__global T *g_idata, __global T *g_odata, int n) {
2      unsigned int tid0 = (get_local_id(0) / 32) * 32 * 2 + get_local_id(0) % 32;
3      unsigned int tid1 = tid0 + 32;
4      unsigned int i0 = get_group_id(0) * 2 * get_local_size(0) + tid0;
5      unsigned int i1 = get_group_id(0) * 2 * get_local_size(0) + tid1;
6      __local T sdata[BLOCK_SIZE];
7      sdata[tid0] = i0 < n ? g_idata[i0] : 0;
8      sdata[tid1] = i1 < n ? g_idata[i1] : 0;
9      barrier(CLK_LOCAL_MEM_FENCE);
10     for (unsigned int s = 2*get_local_size(0)/2; s>0; s>=1) {
11         if (tid0 < s) {
12             sdata[tid0] = sdata[tid0]+sdata[tid0+s];
13         }
14         if (tid1 < s) {
15             sdata[tid1] = sdata[tid1]+sdata[tid1+s];
16         }
17         barrier(CLK_LOCAL_MEM_FENCE);
18     }
19     // only one write - the condition of the second will never be true
20     if (tid0==0) g_odata[get_group_id(0)] = sdata[0];
21     if (tid1==0) g_odata[get_group_id(0)] = sdata[0];
22 }

```

```

1  __kernel void block_coarsened(__global T *g_idata, __global T *g_odata, int n) {
2      unsigned int tid = get_local_id(0);
3      unsigned int i0 = 2 * get_group_id(0) * get_local_size(0) + get_local_id(0);
4      unsigned int i1 = (2 * get_group_id(0) + 1) * get_local_size(0) + get_local_id(0);
5      __local T sdata0[BLOCK_SIZE];
6      __local T sdata1[BLOCK_SIZE];
7      sdata0[tid] = (i0 < n) ? g_idata[i0] : 0;
8      sdata1[tid] = (i1 < n) ? g_idata[i1] : 0;
9      barrier(CLK_LOCAL_MEM_FENCE);
10     for(unsigned int s=get_local_size(0)/2; s>0; s>=1) {
11         if (tid < s) {
12             sdata0[tid] += sdata[tid + s];
13             sdata1[tid] += sdata[tid + s];
14         }
15         barrier(CLK_LOCAL_MEM_FENCE);
16     }
17     if (tid == 0) {
18         g_odata[2 * get_group_id(0)] = sdata0[0];
19         g_odata[2 * get_group_id(0) + 1] = sdata1[0];
20     }
21 }

```

Fig. 3. A reduction kernel from the NVidia OpenCL SDK shown uncoarsened, with thread-level, and block-level coarsening, for stride $S = 32$ (thread-level coarsening only) and coarsening factor $C = 2$.

3.2 Block-Level Coarsening

With block-level coarsening, the number of threads per block remains unchanged so that the number of executed thread blocks is reduced by the coarsening factor. Because each thread block has to handle an increased workload, resource requirements per block, in terms of register and shared memory usage, will typically increase, as with thread-level coarsening.

The stride parameter now determines which blocks will be merged. Unlike thread-level coarsening, the stride has no influence on memory coalescing, as the original memory access patterns of the uncoarsened code within each block are preserved. Choosing a different stride gives a degree of control over the order in which blocks are scheduled. This is in line with NVidia's CUDA and OpenCL programming guidelines, which state that the thread block scheduler does not commit to schedule blocks in any particular order (see NVidia (2010)). In practice, we have not encountered any significant correlation between stride parameter and performance and therefore choose a static value of 1 for the stride throughout our evaluation.

In the kernel body, code duplication is based on calls to `get_group_id()`, and includes all dependent instructions (cf. Table 1).

Shared memory is allocated on a per-block basis. As the work of several thread blocks is combined into one, shared memory regions need to be duplicated along with other instructions of the kernel body. However, a shared memory region needs to be duplicated if and only if it handles a value that is also duplicated in the process of thread coarsening. For dynamically allocated shared memory regions passed as function parameters to the kernel, such duplication is not possible. Throughout our experiments, we have thus chosen to restructure dynamically allocated shared memory into static allocations as a manual step, as can be seen in Figure 3, where shared memory pointers were originally passed in as function arguments.

The third code block of Figure 3 shows the coarsened version (`block_coarsened`) of the original code (first code block in the figure), this time built using the block-level coarsening rules of Table 1. The example shows that after rewriting the call to `get_global_id(0)` using its definition, all instances of `get_group_id(0)` and dependent instructions are duplicated. Note that this includes the allocated shared memory region, giving each of the two merged blocks their own memory region. Unlike thread-level coarsening, no code region requires duplication (this characteristic is more important for longer-running code regions). As such, if the duplicated code regions were expensive to execute, the performance achieved by the two coarsening modes could be expected to differ significantly from each other. The kernel will be executed by half the number of thread blocks, with each block performing twice the number of accesses to global memory as the original.

3.3 Preserving Block Semantics

It is important to ensure that coarsening is a sound transformation in that the semantics of each thread block are preserved. By this, we simply mean that the output of a block is the same before and after coarsening to preserve the way a kernel communicates with other kernels or host code.

To illustrate this, consider the reduction kernel shown in Figure 3, where each block computes one value of a list of partial sums. The transformation is sound if the produced list is unaltered, i.e., if each block produces the same output value to the same list index as before. In the particular case of a reduction operation, this rule could potentially be relaxed if the intention is to perform a full reduction, but this is not generally possible in other cases.

Block semantics are preserved by modifying the launch configuration in a way that corresponds to the applied code transformation rules stated above. Consider again the examples in Figure 3. Both versions should produce the same unaltered list of partial sums as the uncoarsened code. With thread-level coarsening applied, the same single value as before should be written to its


```

1  kernel void memcpy(global float* input, global float* output, int size) {
2      size_t globalId = get_global_id(0);
3
4      for (unsigned int index = 0; index < size; ++index) {
5          output[globalId * size + index] = input[globalId * size + index];
6      }
7  }
8
9  kernel void memcpy_coarsened(global float* input, global float* output, int size) {
10     size_t globalId0 = 2 * get_group_id(0) * get_local_size() + get_local_id();
11     size_t globalId1 = (2 * get_group_id(0) + 1) * get_local_size() + get_local_id();
12
13     for (unsigned int index = 0; index < size; ++index) {
14         output[globalId0 * size + index] = input[globalId0 * size + index];
15         output[globalId1 * size + index] = input[globalId1 * size + index];
16     }
17 }

```

Fig. 4. Memcpy kernel.

intended index, while under block-level coarsening two “merged” blocks should output two values each to their respective indices. However, this is the case if and only if for thread-level coarsening (coarsening factor 2) the number of threads per block is halved while the total number of blocks is retained, while under block-level coarsening the thread block size is retained while the total number of blocks is halved. This is also reflected in the above code-transformation rules. This means that thread block sizes cannot be chosen freely during coarsening, as Magni et al. (2013) suggest. A notable exception are kernels that do not refer to any block semantics in their implementation, such as many linear algebra kernels, or the examples shown in Figures 4 and 5 below, which allow free manipulation of thread block sizes, in line with Magni et al. (2013). The underlying issue is that the CUDA and OpenCL programming models allow writing a kernel body both from the perspective of an individual thread (cf. Figure 4) and from the perspective of a thread block (cf. Figure 3) (or a mixture thereof). The latter, however, is the more general and should be followed in discussions involving code transformations.

4 CACHE PRESSURE

For some kernels, coarsening can significantly increase pressure on the cache, as illustrated by the example in Figure 4. The implementation copies a matrix, using one thread to copy an entire row and each thread loops over adjacent elements belonging to the same cache line before moving on to the next. This is non-optimal for a GPU, but it serves to illustrate the effect. The number of cache lines being accessed at any time is the same as the number of simultaneously active threads. If we coarsen the kernel, then, in the worst case, the same total number of threads will be executing simultaneously as in the original kernel, but with each thread doing more work. In this worst case, the number of cache lines being accessed at any time will thus scale with the coarsening factor and this will increase the pressure on the cache. In practice, since higher coarsening factors can yield lower occupancy and result in fewer active threads, the effect is not always linear. However, in the presence of cache line re-use, the cache pressure frequently increases with the coarsening factor and that this typically outweighs the benefits that coarsening might otherwise bring. Interestingly, the issue here is cache line re-use, which is generally seen as a good thing for performance on conventional architectures, but which is a potential problem when thread coarsening for a GPU. Cache pressure is, however, not an issue when data is accessed in a streaming manner, where there is no data re-use or cache line re-use.

The problem is not just restricted to GPU kernels containing loops, but can also happen in simultaneously executing instances of a single program statement. As an example, Figure 5 illustrates

Fig. 5. Matrix transpose kernel.

a matrix transpose kernel based on the NVidia OpenCL SDK. Here each thread reads a single element from a source array at a given position (x, y) and writes it to a target array at its transposed index (y, x) (cf. Micikevicius (2012)). Cache lines are read in a coalesced fashion by threads of single warps (indicated by colour), but written to differing cache lines in an uncoalesced way. Data that is cached as a result of a read will not be accessed again for the remainder of the program. However, writes to a cache line come from multiple warps, so the line should ideally not be evicted until it has been written to by each of those warps.

How, then, does coarsening affect the behaviour of this kernel with regard to cache pressure? The answer depends on how individual threads are scheduled on the SM and how outstanding writes are handled by the memory system. If, for example, we coarsen by a factor of two then in the worst case twice as many read and write instructions may be in flight at any time, as there will be two instances of the code on line 12 of Figure 5(a). The reasoning is similar to the above case. With more write instructions in flight the average time taken to complete the writes to a given cache line will increase² and this will increase pressure on the cache.

In practice, however, many kernels do not re-use cache lines in this way. For instance, if the above example were rewritten to exploit coalesced memory accesses, data from an entire cache line will be read or written in a single memory access and the data held in the cache line evicted.

4.1 Re-Use Distance Analysis

The problem of determining whether a kernel performs cache line re-use in the above sense can in theory be solved by applying re-use distance analysis, which is a well-known technique for analysing cache locality (Ding and Zhong 2003; Wolf and Lam 1991). Knowing the exact distance of a re-use makes it possible to predict whether a given memory access is likely to be served from cache, or is likely to trigger a cache miss. The former scenario poses a potential performance risk when applying thread coarsening, as coarsening effectively increases re-use distances, which in turn accounts for increased cache miss rates.

Re-use distance analysis requires an ordered memory access trace, making it not directly suitable for GPUs due to the unknown warp schedule. In practice, this can be circumvented by making assumptions on the order in which warps are scheduled. The approach requires that a substantial number of warps be analysed to get an accurate prediction of the distance. One problem here is that the applied thread block dimensions are not actually known until runtime. The more significant problem, however, is that the approach is typically expensive in both memory footprint and runtime. For instance, Nugteren et al. (2014) report a memory footprint of typically around 2GB with an execution time of around 10 seconds runtime (plus 4 minutes additional time for emulation in Ocelot (Diamos et al. 2010)) for the cutcp benchmark.

We require an analysis that is able to execute at program runtime (i.e., kernel compile-time). This is to integrate with our coarsening pipeline (see Section 5.1), which is implemented as part

²For a machine in the steady state, this would be a consequence of Little's Law.

of the OpenCL run-time API, and can thus, for instance, analyse kernels that were generated at runtime.

4.2 Approximate Cache Line Re-Use Analysis

Because we are only interested in *whether* a kernel re-uses cache lines, we do not require a full re-use distance analysis. Instead, we analyse solely for re-use, regardless of the distance.

To do this, we implement an LLVM pass that performs partial symbolic execution of a given kernel, with a view to computing the access locations associated with pointer and array accesses to global memory. These are the only accesses handled by the cache hierarchy. Kernel implementations typically feature separate sets of instructions dealing with memory locations on the one hand, and data stored at these locations on the other. These data-related instructions are disregarded by our analysis. Location-related instructions are based on integer arithmetic, and are therefore frequently computationally cheaper than data-related instructions. The first stage of the analysis performs a pass over the LLVM IR to identify the set of instructions associated with calculating access locations of global memory accesses. A separate pass is then performed over these instructions, which implements the rules described below.

Across our target platforms, thread blocks may assume maximum dimensions of up to 1,024 threads in a space of $1024 \times 1024 \times 64$ threads. However, our symbolic analysis assumes a single thread block of 32, 32×2 , or $32 \times 2 \times 2$ threads for one-/two-/three-dimensional kernels, respectively, as the exact launch configuration is not known when the analysis is run. As a result of that, the memory footprint and runtime of the analysis is thus effectively kept low. (Note that the low number of simulated threads suffices to analyse for a simple yes/no answer, whereas more threads would be needed to obtain an accurate distance measure.)

The idea is to analyse accesses to global memory by constructing Memory Access Descriptors (MADs) that determine at which offset each thread would access a given pointer or array variable. MADs are implemented as matrices of up to three dimensions, with each entry in the matrix corresponding to one particular thread of a single simulated thread block. MADs assume the dimensions encountered in the corresponding arithmetic expressions. As an example, the matrix transpose kernel in Figure 5 has two-dimensional MADs, while one dimension is sufficient to capture accesses in the memcpy kernel of Figure 4.

MADs are constructed by performing a pass over the previously identified location-related instructions described above. When processing the LLVM IR the following cases apply:

- *OpenCL Functions*: For calls to OpenCL built-in functions (such as `get_global_id()` or `get_local_size()`, and the like), we construct a new MAD containing sample ids. To do this, we assign consecutive global ids starting with 0. Corresponding values for all other OpenCL functions can be derived and set accordingly, also with respect to thread block dimensions described above. Recall, that intention for the analysis of a single thread block is to yield insight into the memory access patterns of any thread block, whichever ids we choose.
- *Variables, Function Parameters*: Variables whose values are unknown are substituted by arbitrarily chosen sample values, making sure that different values are used for different variables. This also applies to function parameters. Naturally, for any variable with an existing definition the predefined value will be used.
- *Arithmetic Operations*: The arithmetic operations considered correspond to array index calculations, e.g., $A[i * n]$. In this case, we simulate the corresponding operation, retrieving the existing MADs for the operands, e.g., i and n in the example. The resulting MAD has the combined dimensions of the operand MADs, to correctly represent the captured expression (see above).

- *Conditionals*: For conditionals, both branches are evaluated separately from each other. Thus, if two branches construct different versions of the same MAD, all versions are retained and used separately for analysing memory accesses. MADs that are jointly constructed by different branches in the case of thread divergence are not considered in the current version of the analysis. The benefit is that all branches are analysed regardless of their entry condition. The disadvantage is that cache line re-use which occurs *only* in the presence of thread divergence and does not occur otherwise, is currently not detected. However, this appears to constitute a relatively uncommon scenario—it is not encountered in any of our benchmarks, for example.
- *Loops*: For loops, the goal is to detect cache line re-use both within the loop body and between iterations. We found in our experiments that this can be captured surprisingly accurately for a large number of loops by analysing only two adjacent iterations—a number which is naturally compounded for nested loops. If, for instance, the loop nest contains a conditional expression causing it to apply cache line re-use only in later iterations, this will be detected by the fact that the analysis considers all branches. Our analysis will not detect cache line re-use if the re-use occurs in non-adjacent iterations of a loop (in practice, such a scenario of cache line re-use increases the re-use distance and hence the likelihood that a cache eviction will have occurred, rendering the result of the analysis less important). We do not require loops to have affine array accesses (cf. Bastoul (2004)), but support a variety of loop structures, so long as there is an induction variable.
- *Memory Accesses*: When a memory access is encountered, the existing MADs are used to test for potential cache line re-use. If multiple versions of the MAD exist (e.g., after a conditional), these are tested independently of each other. Each MAD is processed in chunks equal to the warp size (i.e., 32 threads). For each warp, the address offsets are translated into cache line offsets, taking cache line width and alignment of the memory access into consideration. Duplicates within a warp are treated as a single coalesced access. A memory access is considered to *not* be performing cache line re-use, if:
 - The cache line offsets accessed by any warp have not been accessed by a previously simulated warp within the same memory access instruction.
 - The cache line offsets accessed by any warp have not been accessed in a previous memory access instruction to the same symbol. (The assumption is that pointer arguments to a given kernel are not aliases of one another; this is typically the case in OpenCL kernels, although the property is rarely declared explicitly; note that propagating this assumption would incur unwanted side effects).
 - All warps access the same cache line within a single instruction (this case is ignored due to its small impact).
 - The memory access is fully coalesced and is a store operation (this case would not trigger a cache miss).

Note that the analysis can simply “opt out” if it encounters any unsupported feature or loop structure. In this case, coarsening will not be applied to the kernel.

Data-dependent memory accesses (typically of the form $A[B[i]]$) constitute a special case that cannot be handled by this analysis, as they require either full simulated execution or a dynamic approach based on actual program execution.

The analysis will terminate as soon as any cache line re-use or data dependency is detected. If the analysis completes without detecting either of these scenarios, the analysed kernel is considered to be safe to coarsen.

4.3 Comparison with Re-Use Distance Analysis

Compared with re-use distance analysis, the order of accesses is now unimportant, making it unnecessary to consider specific warp schedules. The analysis can be executed with a small number of warps, as no “filler” warps are required to accurately determine the distance between two accesses. Finally, no caching model, including knowledge of the hierarchy or replacement policy, is required. Instead, it suffices to remember which cache lines are requested.

However, the approach surrenders some of the accuracy captured in the re-use distance metric. For instance, if the re-use distance is sufficiently large, it can be reasoned with the help of a cache model, that the accessed data will have been evicted and is guaranteed to cause a miss. A model that is unaware of distance between accesses is not able to identify this. However, Wang and Xiao (2016) claim that many kernels perform memory accesses either in a streamed manner (accessing data only once, rendering caching redundant), or typically exhibit short re-use distances. This could, in practice, be beneficial to our approach, as it lacks accuracy in particular for long re-use distances, where it is not able to predict that an eviction has taken place.

Evaluating our approach on 19 kernels from the Rodinia Benchmark Suite (Section 6), the maximum memory requirement was measured to be around 4KB. This is in addition to any memory requirements of the llvm opt tool, which typically requires 50–100MB. The total execution times (incl. opt) ranged from 0.01 to 0.1 seconds, an improvement of several orders of magnitude both in terms of runtime and memory footprint compared to full re-use distance analysis (Nugteren et al. 2014) (cf. 2GB memory footprint with an execution time of around 10 seconds runtime plus 4 minutes additional time for emulation in Ocelot).

5 PREDICTING OPTIMAL COARSENING FACTORS

To determine whether to coarsen a given kernel we use the analysis described above to determine whether or not a kernel re-uses data that is mapped to the same cache line. The assumption we make in such cases is that coarsening is likely to reduce, rather than increase, performance and this is backed up by observation (see Section 6). In these cases, we choose not to coarsen the kernel and in all other cases we estimate the optimal coarsening factor based on thread occupancy metrics.

The optimal coarsening factor is estimated to be the largest factor that preserves the occupancy of the original kernel. Beyond this the occupancy will start to drop, as more resources are required to support increasingly larger kernels, and our current assumption is that this will lead to reduced performance. This is one sense in which our model is approximate, as it is well known that some kernels can exhibit better performance at lower occupancy (Volkov 2010), an effect that we observe in some of our benchmarks. One of our objectives is to evaluate how well the simple occupancy-based metric works in practice.

5.1 Implementation

We have extended the existing thread coarsening pipeline presented by Magni et al. (2013) by adding support for block-level coarsening, following the specification given in Section 3. An LLVM pass to perform cache-line reuse analysis was also added (see Figure 6). The final step is to estimate the optimal coarsening factor for kernels that are deemed not to be cache dependent. The system will then execute the kernel with the automatically selected coarsening factor, although a coarsening factor may also be specified manually if desired.

The existing code described in Magni et al. (2013) implements thread-level coarsening as a source-to-source transformation and this is realised by intercepting calls to the OpenCL runtime API for compiling and launching a kernel.

Fig. 6. Coarsening pipeline.

Intercepting calls to `clBuildProgram()`, which handles compilation of a kernel, allows that kernel to be parsed into LLVM IR via Clang, running the coarsening pass, which is implemented as an LLVM optimisation pass, and translating the result back to OpenCL using AXTOR, an abstract syntax tree extractor.

Intercepting calls to `enqueueNDRangeKernel()`, which handles the launching of a kernel, is necessary because it is only at this stage that the exact launch configuration is known. The requested number of thread blocks and the size of each thread block, are adjusted depending on the specified coarsening factor, as discussed above in Section 3.

Intercepted calls to `clBuildProgram()` are extended by invoking the cache dependence analysis described above. Subsequently, the kernel is statically compiled multiple times with different coarsening factors. The additional compilation cost depends on the number of coarsening factors considered (we use only powers of two but this is an arbitrary decision). Importantly, since the optimal coarsening factor depends on the specified problem size, the choice of coarsening factor can only be made when the launch configuration is known, i.e., when `enqueueNDRangeKernel()` is called.

Compiling differently coarsened versions of the same kernel with the compiler’s verbosity flag switched on allows us to extract the compiler output by querying `clGetProgramBuildInfo()`. For NVidia compilers, the build log contains easily parsable information for each instance of the code including register usage per thread and statically allocated shared memory usage per thread block. Note, however, that the latter figure needs to be interpreted in the context of the thread block size, which is not known until `enqueueNDRangeKernel()` is called. Information extracted from the build log is cached separately alongside the kernel code.

When `enqueueNDRangeKernel()` is called to execute a kernel with a given launch configuration, the occupancy achievable by each of the coarsened instances of a given kernel is calculated in terms of the number of requested thread blocks, number of threads per block, registers per thread, and the amount shared memory per thread block. Architectural parameters are also required, such as the available shared memory per SM, maximum executable thread blocks per SM, registers per SM, maximum active threads per SM, and so on, and may be provided as environment variables. The occupancy calculation is described in the NVidia SDK documentation.

6 EVALUATION

The objective of our evaluation is to answer the following questions about thread coarsening policy:

Table 2. Properties of the Test Systems

Architecture	NVidia Kepler	NVidia Maxwell	NVidia Pascal
Device Name	GeForce GTX TITAN Black	GeForce GTX TITAN X	GeForce GTX 1080
CUDA Compute Capability	3.5	5.2	6.1
OpenCL Version	1.2	1.2	1.2
CPU	Intel Core i7-4770K	Intel Core i7-4770K	Intel Xeon E5-1630 v3
OS	Ubuntu 14.04	Ubuntu 14.04	Ubuntu 16.04
Streaming Multiprocessors (SM)	15	24	20
Max. active threads per SM	2048	2048	2048
Max. thread blocks per SM	16	32	32
Registers per SM	64k	64k	64k
Shared Memory per SM	48kB	96kB	96kB
Max. Shared Memory per block	48kB	48kB	48kB

- How useful is occupancy as a guiding principle for selecting good coarsening factors?
- Does our framework ever reduce the performance of an uncoarsened kernel, i.e., is any kernel being coarsened when it should not?
- How does the performance of block-level coarsening compare with that of thread-level coarsening?

As such, we are not primarily concerned with re-evaluating potential performance benefits of thread coarsening, which others have done before (see Section 7).

We evaluate our findings on two sets of benchmarks, namely the reduction kernels from the NVidia OpenCL SDK and a selection of kernels from the Rodinia Benchmark Suite. The NVidia OpenCL SDK provides seven versions of a reduction kernel, each manually optimised to a different degree. By choosing these kernels to be part of our evaluation, we are able to investigate the efficiency of our model under different degrees of utilised peak performance. In addition, we have chosen the Rodinia Benchmark Suite for our evaluation to test our model and cache line re-use analysis on a wider variety of kernels from different problem domains.

It should be noted, that the Rodinia OpenCL kernels do not implement any vendor-specific optimisation, as they are targeted at a variety of platforms, and should hence be viewed as unoptimised code. This is unlike the reduction kernels, which are provided in hardware vendor’s SDK and implement different levels of optimisation as outlined below (see Table 3). Our experimental setup is described in Table 2.

6.1 Reduction Benchmarks

The NVidia OpenCL SDK includes seven reduction kernels that are manually optimised to different degrees (see Harris (2007)). Table 3 outlines the optimisations applied to each version of the code. We measured the throughput of each kernel for each architecture and each mode of coarsening using a test data set of 450MB of single-precision floating-point values.

Applying cache-line reuse analysis to these kernels suggests that all versions are suitable for coarsening. Performance graphs for each architecture and coarsening mode can be found in Figure 7, each graph plotting the applied coarsening factor against the achieved performance for each of the seven reduction kernels. The results shown are averages from 10 independent executions of each kernel for each coarsening factor.

The coarsening factors selected are highlighted with a cross. The model predictions for `reduce0` to `reduce5` turn out to be identical. This is due to the fact that all versions of the code request to use

Table 3. Optimisation Levels of Reduction Kernels from the NVidia OpenCL SDK

Kernel	Optimisation
reduce0	Interleaved addressing (using modulo arithmetic) with divergent branching.
reduce1	Interleaved addressing (using contiguous threads) with bank conflicts.
reduce2	Sequential addressing, no divergence or bank conflicts.
reduce3	Uses $n/2$ threads, performs first level during global load.
reduce4	Unrolled loop for last warp, intra-warp synchronisation barriers removed.
reduce5	Completely unrolled, using template parameter to assert whether the number of threads is a power of two.
reduce6	Multiple elements per thread, small constant number of thread blocks launched. Requires very few synchronisation barriers.

Fig. 7. Reduction benchmarks, plotting achieved throughput over coarsening factor. Circled are the points selected by our model.

the same amount of shared memory, which in this case acts as the limiting factor to occupancy. Therefore, the estimated maximum coarsening factor is similar for all but the last kernel. The optimal coarsening factor is four on the Kepler architecture, and eight on the Maxwell and Pascal architectures, which our model is able to correctly identify in each case.

The graphs show a performance increase that corresponds to a speedup of up to 2.85x for thread-level coarsening (reduce3 on Kepler), and up to 5.08x for block-level coarsening (reduce0 on Pascal). The average performance increase (geometric mean) of all kernels under thread-level and block-level coarsening is 1.73x and 1.97x, respectively.

Table 4. Overview of Rodinia Kernels Used in Our Experiments, Including Thread Block Sizes

Name	Benchmark	Kernel	Dwarfs	Domain	Threads per block
findK	B+ Tree	findK	Graph Traversal	Search	256
findRangeK	B+ Tree	findRangeK	Graph Traversal	Search	256
bpnn_forward	Back Propagation	bpnn_layerforward	Unstructured Grid	Pattern Recognition	16x16
bpnn_adjust	Back Propagation	bpnn_adjust_weights	Unstructured Grid	Pattern Recognition	16x16
bfs1	Breadth-First Search	BFS_1	Graph Traversal	Graph Algorithms	256x1
bfs2	Breadth-First Search	BFS_2	Graph Traversal	Graph Algorithms	256x1
hotspot	HotSpot	hotspot	Structured Grid	Physics Simulation	16x16
hotspot3D	HotSpot 3D	hotspotOpt1	Structured Grid	Physics Simulation	64x4
kmeans	Kmeans	kmeans_kernel_c	Dense Linear Algebra	Data Mining	256
kmeans_swap	Kmeans	kmeans_swap	Dense Linear Algebra	Data Mining	256
lavamd	LavaMD	kernel_gpu_opencl	N-Body	Molecular Dynamics	128
nn	k-Nearest Neighbors	NearestNeighbor	Dense Linear Algebra	Data Mining	256
dynproc	PathFinder	dynproc_kernel	Dynamic Programming	Grid Traversal	256
pf_findIdx	Particle Filter	find_index	Structured Grid	Medical Imaging	512
pf_normalize	Particle Filter	normalize_weights	Structured Grid	Medical Imaging	512
pf_sum	Particle Filter	sum	Structured Grid	Medical Imaging	512
pf_likelihood	Particle Filter	likelihood	Structured Grid	Medical Imaging	512
sc_memset	Streamcluster	memset_kernel	Dense Linear Algebra	Data Mining	256x1
sc_pgain	Streamcluster	pgain_kernel	Dense Linear Algebra	Data Mining	256x1

The graphs show that kernels coarsened according to our model match or exceed the performance of uncoarsened kernels that are optimised to roughly two degrees more—all of which can be achieved in a fully automated fashion.

The reduce6 kernel is launched with a constant number of thread blocks, small enough that all can be actively processed at the same time. This renders coarsening obsolete as applying the optimisation would, in practice, mean a reduction in occupancy. As our model considers the requested launch configuration in its occupancy calculation, this is correctly identified, and the kernel is not coarsened. Without this restriction the kernel would run at 0.26x to 0.57x of its original performance.

All but the last of the reduction kernels may be considered barrier-intensive. Liu et al. (2016) have recently highlighted the importance of optimising stalls at synchronisation barriers. This characteristic seems to account for a significant proportion of the achieved speedup, as synchronisation barriers are either hit by fewer thread blocks (block-level coarsening), or by fewer threads per block (thread-level coarsening); both appear to have a similar effect on performance.

6.2 Rodinia Benchmarks

The Rodinia benchmarks cover various problem domains, inspired by the Berkeley dwarfs (Asanovic et al. 2006). The benchmarks offer the functionality to freely specify the desired problem size and to generate corresponding test data sets.

We select kernels from benchmarks that are easy to scale up in size and which were compilable using the existing coarsening/AXTOR pipeline, after various bug fixes and extensions. This left us with 19 kernels, which are summarised in Table 4.

Figure 8 plots the performance of these kernels for both thread-level and block-level coarsening on each architecture (see Table 2). In each graph, the left axis (solid line) indicates the achieved speedup of the kernel and the right axis (dashed line) the theoretical occupancy of the kernel for each coarsening factor. Occupancies are expressed as a percentage of the peak (100%). The circles mark the coarsening factor that would be chosen in the absence of cache line re-use analysis and the crosses mark the one actually selected, i.e., taking that analysis into account. Note that our

Fig. 8. Rodinia—plotting speedup over coarsening factor (left axis, straight line), against occupancy (right axis, dashed line), with unadjusted model predictions (X mark) and adjusted by CLR analysis (O mark).

model predictions are limited by both theoretical occupancy and problem size, the latter of which is not highlighted in the graphs.

For readability, we have grouped the benchmarks by the outcome of the cache line re-use analysis, as indicated to the right of the diagram: eight kernels analysed as safe to coarsen, five kernels for which cache line re-use has been detected, and six kernels which were analysed to be data-dependent. The decisions of the analysis were straightforward to verify by manually inspecting the code of the benchmarks.

The choice of a constant stride value of $S = 32$ in thread-level coarsening reveals a problem which did not occur with the reduction benchmarks. Rearranging inequality Equation (1) in Section 3.1, we have the constraint that $C \leq \text{get_local_size}(d)/S$. However, many of the Rodinia benchmarks have relatively small thread block sizes (see Table 4) which means that many benchmarks can only be coarsened by small factors; this is reflected in the graphs of Figure 8. We could pick a smaller stride, e.g., $S = 8$. However, this risks breaking memory coalescing, as discussed in Section 3.1. An advantage of block level coarsening is that this problem does not arise.

6.2.1 Kernels Safe to Coarsen. The first group of eight benchmark consists of kernels that our analysis assessed as safe to coarsen, meaning that coarsening should either have a beneficial or at least a neutral effect on kernel performance. The `bpnn_forward` kernel is largely algebraic and of medium complexity, featuring some control flow elements as well as synchronisation barriers. The kernel benefits well from coarsening, gaining between 2.2x and 3.3x speedup in block-level coarsening mode, being able to benefit from shared instructions in the coarsened code as well as having a control flow structure that synthesises well with the coarsening optimisation.

The `nn` kernel is an example of kernel on which coarsening has a neutral effect, which results from a feature in its implementation: All code of the kernel body is wrapped inside an `if(get_global_id(0) < N)` statement, which essentially performs bounds-checking to ensure that the kernel body is not executed by more threads than intended. This is a common feature in GPU codes, which, however, does not synthesise well with the coarsening algorithm. Recall that the coarsening algorithm as described by Magni et al. (2013) duplicates code regions where the entry condition depends on a variable that itself requires duplication. This means, if coarsening is applied with a factor of two, there will be two derivatives of `get_global_id`, and one `if`-statement for each of those, which are processed separately. The coarsened version is thus not able to profit from any of the benefits of thread coarsening (e.g., shared instructions), resulting in a largely neutral performance effect. The `kmeans` kernel follows the same pattern of performing a check on the global ID, although it is not clear why this resulted in a performance decrease in the Pascal experiments. The `pf_sum` kernels exhibits the same pattern, while the `pf_normalize` kernel has several successive code regions each having such conditionals. In both cases the performance effect is neutral as expected.

The `sc_memset` kernel is a particularly trivial kernel, setting each field of an array to the same value. The performance benefit of thread-level coarsening does not match that of block-level coarsening, which is potentially due to a higher overhead that thread-level coarsening incurs. However, the kernel does perform best at a very low occupancy, an effect that may be explained by Volkov (2010).

6.2.2 Kernels with Cache Line Re-Use. The `hotspot3D`, `kmeans_swap`, `pf_likelihood`, and `sc_pgain` kernels were analysed to perform cache line re-use. Across these benchmarks, coarsening (at lower occupancy) yielded a significant speedup for one experiment. While the `hotspot3D` kernel would not be coarsened on the basis of our occupancy analysis, the remaining three benchmarks yielded 0.88x, 0.91x, and 0.83x of their original performance in individual experiments, respectively (assuming model predictions are followed).

Table 5. Average Speedups for the Different Categories of Kernels from the Rodinia Benchmark Suite

Category	Max Speedups (Avg)		Unchecked Model (Avg)		Model Predictions (Avg)	
	thread-level	block-level	thread-level	block-level	thread-level	block-level
Safe to coarsen	1.05x	1.35x	1.03x	1.30x	1.03x	1.30x
Cache line re-use	1.10x	1.17x	0.98x	1.08x	1.00x	1.00x
Data-dependence	1.28x	1.19x	1.22x	1.10x	1.00x	1.00x
Total	1.13x	1.25x	1.07x	1.17x	1.01x	1.12x

Listed are average of maximum speedups, speedups for unchecked and actual model predictions (corresponding to circles and crosses in Figure 8, respectively), for thread-level and block-level coarsening.

The observed performance impacts are not as large as we observed in Figure 5, and in some cases the performance curve remains neutral throughout. What this highlights is that although these kernels are classified as unsafe to coarsen, applying the coarsening optimisation is not guaranteed to be detrimental. Rather, in practice, a kernel might have a complex interplay of benefits and risks of coarsening, such that both a decrease and an increase in performance are possibilities, causing the exact performance impact to be somewhat unpredictable.

A clear example for this is the `bpnn_adjust` kernel. This kernel is a small, algebraic kernel, featuring a synchronisation barrier as well as one conditional, but no loops. It benefits from coarsening, as well as from running under lower occupancy due to its cache line re-use. In this particular case, the reason for this is that a reduction in occupancy eases pressure on the cache in addition to the benefits this kernel experiences from being coarsened. This is also why the unadjusted model predictions fall short of the optimal speedup, e.g., in the case of block-level coarsening on Maxwell 1.72x out of 2.78x.

6.2.3 Data-Dependent Kernels. Kernels with data-dependent memory accesses (i.e., typically with memory access patterns of the form $A[B[x]]$) are by default excluded from coarsening, as their memory access pattern cannot be reconstructed by the cache line re-use analysis. Instead, profiled execution or full simulated execution would be required to decide whether coarsening should be applied—and this step may have to be repeated if any change in the data that controls memory accesses has taken place between successive kernel invocations, effectively making this class of kernels hard to deal with. The experiments show that, for instance, the `findK` and `dynproc` kernels are able to achieve consistently good speedups ranging from 1.05x to 1.9x, averaging a speedup of 1.4x if the model had been followed. In contrast to that, the benchmarks for the `pf_findIdx` kernel show that coarsening does not apply to this kernel, such that naively following the model predictions would yield between 0.82x and 0.93x of the kernel’s original performance. While it may appear that following the model predictions, regardless of the cache line re-use analysis, would result in a performance increase on average, it must be considered that this effect is likely due to chance, also with respect to the relatively small sample size, such that this class of kernels remains essentially unpredictable in terms of the performance impact of coarsening.

6.2.4 Speedups. Table 5 lists averaged results for maximum speedups and speedups achieved by our model for the Rodinia experiments shown in Figure 8. As before, we distinguish between “unchecked” model predictions, which have not been adjusted by the results of the cache line re-use analysis, and the model predictions that take the results of the analysis into account.

For kernels that are safe to coarsen, the model can achieve a 1.30x average speedup of a maximum possible 1.35x speedup for block-level coarsening, while for thread-level coarsening

the realised average speedup is 1.03x out of 1.05x. This reflects the practical difficulties we encountered based on the stride parameter.

Kernels classified as performing cache line re-use averaged a performance of 0.97x and 1.08x for the two coarsening modes, respectively, and were excluded from being coarsened. The latter number is positive only because of the `bpnn_adjust` kernel, without which the speedup for block-level coarsening amounts to 0.95x. On the other hand, kernels deemed safe to coarsen yielded a clear average performance speedup, with the exception of corner cases (e.g., `kmeans`), as discussed above.

6.2.5 Discussion. Our experiments show that thread occupancy is a useful metric to use when determining coarsening factor, but also that it is essential to model the effect that coarsening is likely to have on cacheing. The occupancy metric on it is not sufficient. The analysis we perform is conservative, in the sense that we will only coarsen a kernel if we can establish that the pressure on the cache will not increase by doing so. Of course, it may be that any increase in cache pressure may not have significant detrimental effects on performance and/or that any such effects are, in practice, outweighed by other factors. This can be seen to a small extent in benchmarks like `sc_pgain` and `kmeans_swap` for some configurations. It may also be the case that performance increases, despite a reduction in occupancy, and we can see this in some of the benchmarks, e.g., `hotspot`.

In almost all cases, performance either increases or remains flat when we elect to coarsen a kernel, so we conclude that the combination of occupancy metric and cache line re-use analysis is doing a good job of ensuring that our optimisations are *safe* in that they avoid degrading performance. The only exception in our benchmark suite was *k-means*, as discussed above.

Table 5 shows that block-level coarsening results in better overall performance than thread-level coarsening for the Rodinia benchmarks. However, this was not as clearly the case for the reduction benchmarks (see Section 6.1). This highlights the significant practical difficulty of choosing a suitable stride parameter for small thread block sizes. The problem is amplified, in particular, for multi-dimensional kernels, where coarsening with a higher stride is frequently impossible, depending on the exact thread layout of the block. However, as noted earlier, reducing the stride parameter is not safe in all cases as it may break memory coalescing.

Independent of the stride, applying thread-level coarsening can lead more quickly to occupancy reduction than for block-level coarsening. The reason is that while the number of threads per block is reduced, the number of blocks running on each SM is increased to retain the same occupancy. However, the number of blocks running per SM is limited. In practice, very small thread blocks cannot be coarsened at all using thread-level coarsening without effecting a reduction in occupancy.

The practical issues encountered by block-level coarsening appear to be less significant in comparison. Two points in particular are worth mentioning. First, the coarsening factor must evenly divide the number of thread blocks launched. We anticipate that, in practice, this can be circumvented by pre-compiling different or additional coarsening factors, whereas our study is limited to coarsening factors that are powers of two. Second, block-level coarsening produces fewer, longer-running thread blocks and this can lead to longer “tail effects,” where the last few blocks are running on a small number of SMs. Depending on the total workload, this can reduce the overall device utilisation.

The decision as to which coarsening mode should be applied to a given kernel can also depend on which code regions require duplication, and, as such, the exact structure of the resulting coarsened kernel—which, in many cases, differ for the two coarsening modes. Because of this, we can imagine that, in some cases, it may be desirable to choose among the two options based on static analysis of the kernel being compiled.

7 RELATED WORK

Thread Coarsening. Applying thread coarsening to GPU kernels was first discussed as a manual optimisation technique by Volkov and Demmel (2008). Stratton et al. (2012) investigate it as part of a comparison between and investigation into different GPU code optimisations. Unkule et al. (2012) provide a more focused study and the first evaluation of thread coarsening, and present an implementation of the thread coarsening code transformation based on annotated CUDA code. Magni et al. (2013) present a more refined implementation of a thread coarsening algorithm with improved handling of control flow as an OpenCL source-to-source transformation, and include a cross-platform evaluation of the optimisation.

Selecting Coarsening Factors. Unkule et al. (2012) implement thread-coarsening as a semi-automatic optimisation. They mention an experimental heuristic to automatically select coarsening factors, yet implementation details or an evaluation are not provided. They note that in one of their experiments, the maximum speedup is achieved at the lowest occupancy, and hence emphasize the importance of factors other than occupancy for selecting good coarsening factors. Magni et al. (2013) comment that they view the authors' evaluation on a set of five kernels, to which coarsening generally applies, as limited. Magni et al. (2013) consider 17 kernels across five architectures, leading the authors to say that developing an analytical model is "unfeasible" due to the large number of architectures and runtime libraries involved. They present an analysis based on regression trees to characterise the significance of different hardware counters on each platform for achieving good performance when applying coarsening.

The authors follow up their work recommending an auto-tuning-based approach for estimating optimal coarsening factors (Magni et al. 2014b), for which they optimise the size of the required test data set. Finally, they also present a study of applying machine learning techniques to selecting good coarsening factors (see Magni et al. (2014a)).

Code Optimisation. To address the problem of performance portability of CUDA and OpenCL, various approaches have been explored for generating optimised code from higher level languages, for instance see Mainland and Morrisett (2010), or Chakravarty et al. (2011) for a template-based approach. Others presented annotation-based memory management optimisation tools (Ueng et al. 2008) and dynamic optimisation frameworks (Diamos et al. 2010). Others have focused on studying code optimisations in the context of domain-specific languages (DSLs), e.g., Sujeeth et al. (2014).

Auto-tuning as a way to experimentally explore the optimisation space has been long established when it is not clear whether to apply an optimisation and what its parameters should be Demmel et al. (2005). It has been used to target GPGPUs from a high-level language (Grauer-Gray et al. 2012), to generate optimised DSL kernels (Shin et al. 2010), to determine algorithmic choice on heterogeneous architectures (Phothilimthana et al. 2013), and more. To optimise the process of auto-tuning especially in the context of large optimisation spaces, Ryoo et al. (2008) proposed optimisation space pruning, while Magni et al. (2014b) investigate the optimisation of individual experiments to achieve results scalable to larger data sets (interestingly, in the context of thread coarsening). Williams et al. (2011) propose hierarchical and distributed auto-tuning.

Heuristics and cost models have been employed in contexts ranging from compile-time source-to-source transformations (Bondhugula et al. 2008) to runtime code-generation (Luporini et al. 2014), in the latter case specifically to avoid the runtime overhead incurred by auto-tuning. Wen et al. (2014) use a combination of runtime data and code analysis to schedule tasks on heterogeneous CPU/GPU platforms. Huang and Li (2017) use performance models to predict the performance impact of data based in different types of memory on a GPU. Interestingly, Chen et al. (2014) present a framework for data placement on GPUs that utilises a reuse distance-based model.

Others apply machine learning techniques (Fursin et al. 2011), also in the context of thread-coarsening (Magni et al. 2014a).

Program Analysis. In the context of cache line re-use analysis, polyhedral compilation techniques (Bastoul 2004) offer an exact rather than heuristic approach to analysing memory access patterns in loop nests. However, at the time of testing we encountered that too many of our test cases were still unsupported by Grosser et al. (2011), as the tool was still under active development. We leave the integration of polyhedral compilation techniques for future work.

Re-use distance analysis (Wolf and Lam 1991) was first studied on GPUs by Shojania and Li (2009) in an analysis based on GPGPU-Sim (Bakhoda et al. 2009). Nugteren et al. (2014) present an implementation of re-use distance analysis with a cache miss analysis model for GPUs (also see Section 4).

8 FUTURE WORK

The work presented here is, by design, specific to NVidia GPU architectures and exploits information extracted from the NVidia compiler. The idea of making coarsening decisions on the basis of resource usage could, in principle, be adapted to other platforms, such as those of AMD and ARM. Although there is no direct corresponding notion of thread occupancy on such platforms, other resource utilisation measures may prove to be applicable and it will be interesting to explore this further.

A key remaining challenge is to reason about kernel behaviour at lower occupancy. It is well known that peak performance of some kernels occurs at occupancies well below 100% (Volkov 2010); the problem becomes one of detecting the characteristics of such kernels and then determining a suitable reduced occupancy metric.

The correctness of our transformations is also an interesting issue. Given a formal semantics for the OpenCL (or similar) language it seems entirely feasible to prove that our transformations preserve the meaning of the program by formulating, and proving, a suitable correspondence theorem. The somewhat subtle issue of block semantics (Section 3.3) would be interesting to capture, although this lies well beyond the objectives of the present article.

There are clearly improvements that can be made to the compilation pipeline. For example, having compiled a kernel for a particular configuration and coarsening factor, we could cache the result to avoid re-compiling the same configuration at a later time. With that said, the low cost of our analysis means that dynamic compilation of kernels, which is the standard model in the OpenCL API, is completely tractable.

The current implementation could be extended by the ability to coarsen in multiple dimensions; the selection of the stride parameter or the dimension in which coarsening is applied could also be automated, as well as the selection of coarsening mode. The latter could be achieved by identifying which code regions of a given kernel will be duplicated in either coarsening mode, and comparing the resulting shared instructions before applying the transformation.

In the context of cache line re-use analysis, exploring the adaptation of polyhedral analysis may likely prove beneficial for understanding loop nests. In its current form, the analysis does not support thread divergence. The existing analysis could be combined with a benefit analysis, e.g., based on a simple cost-model, to improve accuracy of which kernels should be excluded from coarsening.

9 CONCLUSION

This article has shown that occupancy can be a useful guiding measure for estimating optimal coarsening factors, but only when used in conjunction with cache line re-use analysis. Without

this, it is easy to slow a kernel down, often significantly, because of the effect that coarsening can have on cache pressure. Our approximate cache line re-use analysis has proven to be surprisingly good at distinguishing kernels that are likely to be slowed down by coarsening in this way, although it cannot reason about the exact performance impact that coarsening will have on a given kernel. The analysis is extremely efficient both in terms of execution time and memory footprint when compared with more detailed analyses, e.g., those based on re-use distance. We believe it strikes a good balance between accuracy and efficiency and it will be interesting to see whether it can be applied, or adapted, in other contexts.

We have explored two coarsening strategies, thread level and block level. A comparison of the two has shown that it can be difficult to determine a suitable value for the stride parameter that is associated with thread-level coarsening. Selecting a low stride risks breaking memory coalescing, while selecting a higher value can limit the choice of coarsening factors, especially when dealing with smaller or multi-dimensional thread blocks. As such, for those Rodinia kernels we deem safe to coarsen, block-level coarsening achieved a 1.30x (of a maximum of 1.35x) speedup. In comparison, thread-level coarsening achieved only a 1.03x speedup for the same kernels. However, for the NVidia reduction benchmarks, where choosing a suitable stride parameter was not an issue, the achieved performance gains were more similar, yielding 1.73x and 1.97x for thread-level and block-level coarsening, respectively. On balance, our experiments appear to suggest that block-level coarsening is the preferred option: its performance is generally superior and it is easier to automate, as it does not require the compiler, or user, to select a suitable stride parameter.

REFERENCES

- George Almási, Calin Caşcaval, and David A. Padua. 2002. Calculating stack distances efficiently. In *ACM Sigplan Notices*, Vol. 38. ACM, 37–43.
- Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and others. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.
- Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2009 (ISPASS'09)*. IEEE, 163–174.
- Cedric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 7–16.
- Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 101–113.
- Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming*. ACM, 3–14.
- Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 88–100.
- Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Katherine Yelick. 2005. Self-adapting linear algebra algorithms and software. *Proc. IEEE* 93, 2 (2005), 293–312.
- Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 353–364.
- Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 245–257.
- Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, and others. 2011. Milepost GCC: Machine learning enabled self-tuning compiler. *Int. J. Parallel Program.* 39, 3 (2011), 296–327.
- Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of Innovative Parallel Computing (InPar'12)*, 2012. IEEE, 1–10.

- Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-polyhedral optimization in LLVM. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Vol. 2011.
- Mark Harris. 2007. Optimizing CUDA. *SC07: High Performance Computing With CUDA*. http://gpgpu.org/static/sc2007/SC07_CUDA_5_Optimization_Harris.pdf.
- Yingchao Huang and Dong Li. 2017. Performance modeling for optimal data placement on GPU with heterogeneous memory systems. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing (CLUSTER'17)*. IEEE, 166–177.
- Yuxi Liu, Zhibin Yu, Lieven Eeckhout, Vijay Janapa Reddi, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Chengzhong Xu. 2016. Barrier-aware warp scheduling for throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 42.
- Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. 2014. COFFEE: An optimizing compiler for finite element local assembly. *arXiv:1407.0904*.
- Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014a. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, 455–466.
- Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014b. Exploiting GPU hardware saturation for fast compiler optimization. In *Proceedings of Workshop on General Purpose Processing Using GPUs*. ACM, 99.
- Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 11.
- Geoffrey Mainland and Greg Morrisett. 2010. Nikola: Embedding compiled GPU functions in Haskell. *ACM Sigplan Notices* 45, 11 (2010), 67–78.
- Paulius Micikevicius. 2012. GPU performance analysis and optimization. In *GPU Technology Conference*.
- Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 37–48.
- CUDA NVidia. 2010. Programming guide. (2010).
- NVIDIA Corporation. 2016. *NVIDIA Tesla P100 Whitepaper*. NVIDIA Corporation.
- Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. 2013. Portable performance on heterogeneous architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 431–444.
- Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. 2008. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 195–204.
- Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. 2010. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 253–262.
- H. Shojania and B. Li. 2009. Cache miss analysis for gpu programs based on stack distance profile. In *Proceedings of the IEEE Int'l Conference on Distributed Computing Systems (ICDCS'09)*. 623–634.
- J.A. Stratton, N. Anssari, C. Rodrigues, I-Jui Sung, N. Obeid, Liwen Chang, G. D. Liu, and W. Hwu. 2012. Optimization and architecture effects on GPU computing workload performance. In *Proceedings of Innovative Parallel Computing (InPar'12)*. 1–10.
- Arvind K. Sujeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s (2014), 134.
- Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and W. Hwu Wen-mei. 2008. CUDA-lite: Reducing GPU programming complexity. In *LCPC*, Vol. 8. Springer, 1–15.
- Swapneela Unkule, Christopher Shaltz, and Apan Qasem. 2012. Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In *Proceedings of the 21st International Conference on Compiler Construction (CC'12)*. Springer-Verlag, Berlin, Heidelberg, 21–40.
- Vasily Volkov. 2010. Better Performance at Lower Occupancy. In *Proceedings of the GPU Technology Conference, GTC*. Vol. 10.
- Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2008 (SC'08)*. IEEE, 1–11.
- Dongwei Wang and Weijun Xiao. 2016. A reuse distance based performance analysis on GPU L1 data cache. In *Proceedings of the 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC'16)*. IEEE, 1–8.

- Yuan Wen, Zheng Wang, and Michael F. P. O’Boyle. 2014. Smart multi-task scheduling for openCL programs on CPU/GPU heterogeneous platforms. In *Proceedings of the 2014 21st International Conference on High Performance Computing (HiPC’14)*. IEEE, 1–10.
- Samuel Williams, Leonid Oliker, Jonathan Carter, and John Shalf. 2011. Extracting ultra-scale lattice Boltzmann performance via hierarchical and distributed auto-tuning. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC’11)*. IEEE, 1–12.
- Michael E. Wolf and Monica S. Lam. 1991. A data locality optimizing algorithm. In *ACM Sigplan Notices*, Vol. 26. ACM, 30–44.

Received August 2017; revised February 2018; accepted March 2018