J. Parallel Distrib. Comput. 68 (2008) 1389–1401

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Program optimization carving for GPU computing☆

Shane Ryoo*, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi, Wen-mei W. Hwu

Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 W Main Street, Urbana, IL 61801, United States

ARTICLE INFO

Article history:
Received 11 March 2008
Received in revised form
10 May 2008
Accepted 10 May 2008
Available online 26 June 2008

Keywords:
GPU computing
Parallel computing
Optimization space exploration

ABSTRACT

Contemporary many-core processors such as the GeForce 8800 GTX enable application developers to utilize various levels of parallelism to enhance the performance of their applications. However, iterative optimization for such a system may lead to a local performance maximum, due to the complexity of the system. We propose program optimization carving, a technique that begins with a complete optimization space and prunes it down to a set of configurations that is likely to contain the global maximum. The remaining configurations can then be evaluated to determine the one with the best performance. The technique can reduce the number of configurations to be evaluated by as much as 98% and is successful at finding a near-best configuration. For some applications, we show that this approach is significantly superior to random sampling of the search space.

1. Introduction

Programming for highly-parallel systems has historically been the domain of relatively few experts, with performance tuning done primarily by hand. Because of the relative scarcity of highly parallel applications and the expense of highly parallel systems, there was limited opportunity for exhaustive performance experimentation. Today, however, single-chip, massively parallel systems such as the NVIDIA GeForce 8 Series GPUs are inexpensive and readily available. Unfortunately, the level of effort and expertise required to maximize application performance on these kinds of systems is still quite high. Optimization is further complicated by the sharing of resources at a fine granularity, presenting many choices to application developers of which level(s) of parallelism to utilize. Finally, it is often the case that successive generations of massively-parallel architectures require a complete reapplication of the optimization process to achieve near-maximum performance for the new system.

Optimizing an application for maximum performance on the GeForce 8 Series is not a trivial task. At first glance, it appears to be a multi-variable optimization problem of applying a set of optimization techniques, such as tiling and loop unrolling, to the code. However, the underlying hardware and threading model contain inflexible usage restrictions that affect performance in a non-linear fashion and make the optimization space discontinuous. In addition, the system is complex, with unpredictable behavior resulting from the interaction of various components, particularly the runtime system. A developer also cannot accurately predict the effects of optimizations because the behavior of certain hardware features has not been publicly released. Consequently, the final performance of an optimization configuration is not always readily apparent.

The desire for optimization is significant since the relative performance between configurations can be quite large. For an MRI reconstruction kernel with an optimization space size of 175 configurations, the difference in performance between a hand-optimized implementation and the optimal configuration was 17% and the difference in performance between the worst and optimal configurations was 235%. A full exploration of the optimization space would guarantee finding the configuration with maximum performance, but is generally not feasible because the intent is to run large, compute-intensive applications on these systems.

Rather than attempting to find the best configuration via traditional, phase-ordered compilation, we instead propose an approach that begins with a large optimization space and then prunes it. We term this approach *program optimization carving*. We first develop metrics that capture first-order performance effects as a static analysis. We then prune away parts of the optimization space. In contrast to a full exploration of the optimization space, this methodology eliminates the need to test as much as 98% of the optimization search space. The technique was able to find a configuration with at least 99% of the performance of the maximum within the entire space for several kernels.

☆ This work is based on an earlier work: Program optimization space pruning for a multithreaded GPU, in CGO'08: Proceedings of the sixth IEEE/ACM symposium on code generation and optimization, (c) ACM, 2008. http://doi.acm.org/10.1145/1356058.1356084.

* Corresponding author.
E-mail addresses: sryoo@crhc.uiuc.edu (S. Ryoo), cirodrig@crhc.uiuc.edu (C.I. Rodrigues), ssstone2@crhc.uiuc.edu (S.S. Stone), stratton@crhc.uiuc.edu (J.A. Stratton), ueng@crhc.uiuc.edu (S.-Z. Ueng), bsadeghi@crhc.uiuc.edu (S.S. Baghsorkhi), hwu@crhc.uiuc.edu (W.-m.W. Hwu).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.05.011

Table 1
Properties of GeForce 8800 memories

Memory	Location	Size	Latency	Read-only	Description
Global	Off-chip	768 MB total	200–300 cycles	No	Large DRAM. All data reside here at the beginning of kernel execution. Directly addressable from a kernel using pointers. Backing store for constant and texture memories. Used more efficiently when multiple threads simultaneously access contiguous elements of memory, enabling the hardware to coalesce memory accesses to the same DRAM page.
Shared	On-chip	16 kB per SM	≃register latency	No	Local scratchpad that can be shared among threads in a thread block. Organized into 16 banks.
Constant	On-chip cache	64 kB total	≃register latency	Yes	8 kB cache per SM, with data originally residing in global memory. The 64 kB limit is set by the programming model. Often used for lookup tables. The cache is single-ported, so simultaneous requests within an SM must be to the same address or delays will occur.
Texture	On-chip cache	Up to global	> 100 cycles	Yes	16 kB cache per two SMs, with data originally residing in global memory. Capitalizes on 2D locality. Can perform hardware interpolation and have configurable returned-value behavior at the edges of textures, both of which are useful in certain applications such as video encoders.
Local	Off-chip	Up to global	Same as global	No	Space for register spilling, etc.

We begin by discussing the execution hardware, threading model, and available tools in Section 2. This sets up the various factors that affect performance. Section 3 discusses the most effective optimizations for this architecture, and shows how one needs to consider them to attain the best-performing configuration. Section 4 discusses the program optimization carving technique. Section 5 demonstrates the results achieved with optimization carving, comparing them to random sampling, and how different metrics affect the results. We discuss related work in Section 6 before finishing with our conclusion.

2. Architecture

This work uses the GeForce 8800 GTX GPU[1] as the hardware target for its study. The GeForce 8800 has a large set of processor cores that can directly address a global memory. This general addressibility allows for a more general and flexible programming model than previous generations of GPUs, and allows developers to easily implement data-parallel kernels. In this section we discuss NVIDIA's Compute Unified Device Architecture (CUDA), with emphasis on the features that significantly affect performance. A more complete description can be found in [20,25]. It should be noted that this architecture, although described in more detail than previous GPU architectures, still has details that have not been publicly revealed.

Before discussing the hardware, it is useful to describe the programming and compilation process. The CUDA programming tool kit compiles ANSI C extended with several keywords and constructs. The GPU is treated as a coprocessor that executes data-parallel kernel functions. The user supplies a single source program encompassing both host (CPU) and kernel (GPU) code. These are separated and compiled by NVIDIA's compiler. The host code transfers data to and from the GPU's global memory via API calls. It initiates the kernel code by performing a function call.

2.1. Microarchitecture

Fig. 1 depicts the microarchitecture of the GeForce 8800. The GPU consists of 16 *streaming multiprocessors* (SMs), each containing eight *streaming processors* (SPs), or processor cores, running at 1.35 GHz. Each SP has one 32-bit, single-precision floating-point, multiply-add arithmetic unit that can also perform 32-bit integer arithmetic. Additionally, each SM has two special function units (SFUs) that execute more complex FP operations such as reciprocal square root, sine, and cosine with low latency. The arithmetic units

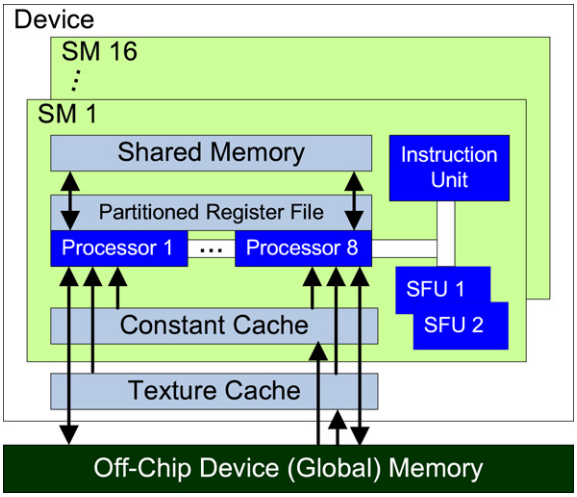


Fig. 1. Organization of the GeForce 8800.

and the SFUs are fully pipelined, yielding 388.8 GFLOPS (16 SM ∗ 18 FLOP/SM ∗ 1.35 GHz) of peak theoretical performance for the GPU.

The GeForce 8800 has 86.4 GB/s of bandwidth to its off-chip, global memory. Nevertheless, with computational resources supporting nearly 400 GFLOPS of performance and each FP instruction operating on up to 12 bytes of source data, applications can easily saturate that bandwidth. In the theoretical worst case, an application running at peak throughput could require up to 2.25 TB/s (1.35 GHz * [128 multiply-add operations * 12 bytes + 32 SFU operations * 4 bytes]) of memory bandwidth, or 26 times the available bandwidth. Therefore, as described in Table 1 and depicted in Fig. 1, the GeForce 8800 has several on-chip memories that can exploit data locality and enable data sharing to reduce an application's demands for off-chip memory bandwidth. For example, each SM has a 16 kB *shared memory* that is useful for data that is either written and reused or shared among threads. For read-only data that is accessed simultaneously by many threads, the *constant* and *texture memories* provide dramatic reduction in memory latency and bandwidth consumption via caching. Accesses to global memory should also be arranged as aligned, contiguous, 16-word lines to achieve an effect called *global memory coalescing*. Any other pattern can achieve only a small fraction of the effective memory bandwidth of coalesced accesses.

Threads executing on the GeForce 8800 are organized into a three-level hierarchy. At the highest level, each kernel creates a single *grid* that consists of many *thread blocks*. The maximum number of threads per block is 512. Each thread block is assigned to a single SM for the duration of its execution. Threads in the

[1] There are several versions of the GeForce 8800 GPU. References to GeForce 8800 are implied to be the GTX model.

Table 2
Constraints of GeForce 8800 and CUDA

Resource or configuration parameter	Limit
Threads per SM	768 threads
Thread blocks per SM	8 blocks
32-bit registers per SM	8192 registers
Shared memory per SM	16 384 bytes
Threads per thread block	512 threads

same thread block can share data through the on-chip shared memory and can perform barrier synchronization by invoking the `__syncthreads` primitive. Threads are otherwise independent, and synchronization across thread blocks can only be safely accomplished by terminating the kernel. Finally, threads within a block are organized into *warps* of 32 threads. Each warp executes in SIMD (single-instruction, multiple-data) fashion, issuing in four cycles on the eight SPs of an SM.

SMs can perform zero-overhead scheduling to interleave warps on an instruction-by-instruction basis to hide the latency of global memory accesses and long-latency arithmetic operations. When one warp stalls, the SM can quickly switch to a ready warp in the same thread block or a ready warp in some other thread block assigned to the SM. The SM stalls only if there are no warps with ready operands available.

2.2. Architectural Interactions

Accurately predicting the effects of one or more compiler optimizations on the performance of a CUDA kernel is often quite difficult, largely because of interactions among the architectural constraints listed in Table 2. Many optimizations that improve the performance of an individual thread tend to increase a thread's resource usage. However, as each thread's resource usage increases, the total number of threads that can occupy an SM decreases. This decrease in thread count can occur in a dramatic fashion because threads are assigned to an SM at the granularity of thread blocks. In short, there is often a tradeoff between the performance of individual threads and the thread-level parallelism (TLP) among all threads.

For example, consider an application that uses 256 threads per block, 10 registers per thread, and 4 kB of shared memory per thread block. This application can schedule 3 thread blocks for a total of 768 threads on each SM. However, an optimization that increases each thread's register usage from 10 to 11 (an increase of only 10%) will decrease the number of blocks per SM from three to two, which decreases the number of threads on an SM by 33%. The GeForce 8800 can only assign two thread blocks (512 threads) to an SM because a third block would increase register usage to 8448, above the 8192 available registers per SM. By contrast, an optimization that increases each thread block's shared memory usage by 1 kB (an increase of 25%) does not decrease the number of blocks per SM. Clearly, the optimization space is inherently non-linear.

2.3. Software tool support

For CUDA compilation, NVIDIA provides a compiler wrapper called *nvcc* that handles all parts of the compilation flow, including linking host and kernel binaries. The compiler also supports several options that programmers can use to debug kernels and to gain intuition on their performance. Two flags are especially useful: `-ptx` and `-cubin`. The amount of time it takes to run nvcc with these flags is much shorter than actual compilation because only the kernel code is processed.

Nvcc compiles kernel code to an assembly-like representation termed *PTX*. This is normally placed in an object file for consumption by the CUDA runtime, which processes this code, performs further optimization such as scheduling, and generates hardware-specific binaries for execution. The `-ptx` flag outputs the PTX in a developer-readable format. Although PTX is not the same code that is executed on the hardware, it often gives insights into why performance degrades or improves after an optimization is applied. In particular, information such as instruction count, instruction mix, and a rough idea of scheduling can be utilized reliably. For example, unrolling a loop with strided memory accesses creates successive operations that operate at different offsets from a base address. PTX shows the savings in address calculations that results from this transformation. Detailed instruction-level scheduling, however, is the domain of the runtime.

The CUDA runtime that generates executable machine code appears to reschedule code and allocate registers. This introduces an uncontrollable element during program optimization and makes the effects of optimizations on local resource usage less predictable. The `-cubin` flag outputs the resource usage of GPU kernel code, including the shared memory used per thread block and registers used per thread. This is critical to understanding the performance of the code because an SM runs the number of thread blocks that fit given their local resource usage. A small change in code can result in resource usage that changes the number of thread blocks executing on an SM, which can significantly impact performance. We use the information provided by `-cubin` to calculate the number of thread blocks that can simultaneously reside on each SM.

3. Optimizations

The basic strategy for achieving good kernel code performance on the GeForce 8800 is to first ensure that executing threads are not starved for data: *global memory bandwidth should not be the limiting factor on performance*. After that, optimizations should balance *high utilization of execution resources* with an *efficient instruction stream*. High utilization, where warps are always available for execution, is accomplished in three ways. First, one can have sequences of independent instructions within a warp so that the same warp can make forward progress. Second, a developer can place many threads in a thread block to improve the likelihood that one warp can execute while others are stalled on long-latency operations, such as memory loads. Third, the hardware can assign up to eight independent thread blocks to an SM. Under high-occupancy conditions, a more efficient program using fewer instructions to complete the same task increases the rate of useful computation performed by the system.

Optimizations rarely improve an aspect of machine-level behavior in an isolated manner. Many optimizations affect several aspects, producing a give-and-take situation between different categories. Moreover, many optimizations increase resource usage and thus compete for a limited budget of registers, threads, and shared memory. The most common way in which optimizations interact and interfere on the GeForce 8800 is by their effects on register usage. For example, an optimization that increases the number of independent instructions after a long-latency instruction generally uses additional registers. This causes register usage of each thread and thread block to increase, which in turn can cause the number of thread blocks assigned to each SM to decrease.

In this section we first discuss the major optimizations for performance. Using a matrix multiplication kernel as an example, we show how these optimizations can be applied to an application to find the best configuration. Fig. 2 illustrates several of these optimizations.

```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
```

(a) Base version.

```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  Ctemp +=
     As[ty][0] * Bs[0][tx];
  ...
  Ctemp +=
     As[ty][15] * Bs[15][tx];

  __syncthreads();
}
C[indexC] = Ctemp;
```

(b) Complete uroll.

```
float a = A[indexA];
float b = B[indexB];
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = a;
  Bs[ty][tx] = b;
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  a = A[indexA];
  b = B[indexB];
  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
```

(c) Prefetching.

```
Ctemp = Dtemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][32];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  Bs[ty][tx+16] = B[indexB+16];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
      Dtemp += As[ty][i]
        * Bs[i][tx + 16];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
C[indexC+16] = Dtemp;
```

(d) 1 × 2 rectangular tiling.

Fig. 2. Matrix multiplication optimization examples code differences from base version are shown in bold.

3.1. Memory bandwidth optimization

The first class of optimizations deals with reducing pressure on memory bandwidth, particularly global memory. The principles of efficiency and utilization matter little to performance if the executing threads are starved for data. Listed here are some techniques for reducing global memory accesses.

- *Capitalize on the reuse of data via local, low-latency memories.* Use of the shared memory and the texture and constant caches is explained in the previous section. This may require restructuring of computation using transformations such as loop interchange [3] to create an amenable access pattern.
- *Improve data locality by coalescing global memory accesses* to fully utilize the width of the global memory bus. Non-coalesced accesses are capable of utilizing only a fraction of the total memory bandwidth. Optimizations include:
  - *Use shared memory as a buffer to load data in bulk*, then access them in whatever pattern the programmer finds convenient from that memory. Even if bank conflicts occur during shared memory accesses, overall performance may improve even without data reuse. Note that memory tiling [31] naturally achieves this effect. It is conceivable that loading data that will not be used may be worth the effects of coalescing, although no such case has been encountered in this study.
  - *Reorganize data and rewrite the application so that the data the kernel accesses are located in aligned, contiguous regions of memory.* One example is to split an array of structures into multiple arrays so that fields that formerly required non-unit strided access are now adjacent in memory. Doing so manually is generally considered poor software engineering practice, but it may be possible for the compiler to perform the transformation or for a smart memory system to reorganize data at runtime. Work by Yamada et al. [32] proposed a combined hardware and software technique to address this issue. Truong et al. [29] recognized the positive cache effects of similar optimizations on different instances of dynamically-allocated objects.

Fig. 2(a) shows a matrix multiplication kernel that consists of 16 × 16 thread blocks and operates on 16 × 16 data tiles. A width of 16 enables global memory coalescing, while the 16 × 16 thread blocks efficiently reuse data. It is possible for applications to still be performance-limited by global memory bandwidth even after these optimizations have been performed. Such applications often have little data reuse and have data organizations that cannot be significantly modified or require major effort to be modified.

3.2. Dynamic instruction reduction

The second category is to reduce the dynamic instruction count per thread, or increase the efficiency of the instruction

stream. Because these are common techniques for improving the performance of applications executing on traditional superscalar processors, their effects are generally well understood. Some of the most effective examples of these optimizations are listed below, along with a short description of their intended effects and common side effects.

- *Common subexpression elimination (CSE).* This optimization removes repeated calculations of an expression, such as an arithmetic operation or the load of a value, and instead uses the saved value of a previous computation from a register. It tends to use additional registers unless a sequence of redundant expressions can be eliminated.
- *Loop-invariant code motion (LICM).* Related to common subexpression elimination, this optimization involves the movement of an expression within a loop to a point outside the loop. This is possible when the sources of the expression do not change during the loop's execution and when the expression is calculated on every path of the loop. It tends to uses additional registers.
- *Loop unrolling.* When a loop has a constant trip count, it is often advantageous to "unroll" the loop by a factor that evenly divides of the trip count, replacing it with a loop that iterates fewer times but performs several times more work per iteration. This has several benefits:
  - Fewer loop iteration instructions are executed.
  - Instructions can be combined. For example, a loop may increment a pointer and then load from the location. When the loop is unrolled, instead of two increment operations, the second load incorporates an offset and a single increment operation is performed. The unrolled matrix multiplication kernel in Fig. 2(b) shows an example of this optimization.

  Complete unrolling of the loop can be very profitable for loops with small bodies since the majority of operations may be loop counter and branch instructions. It also frees up the register that was used as the loop counter. Loop unrolling must be balanced against additional pressure on instruction cache capacity, so in general only the innermost loops of a kernel will be the ones unrolled. It may also trigger other optimizations that change register usage.

3.3. Increasing thread-level parallelism

The goal of the third category of optimization is to improve execution resource utilization by providing enough threads/warps to hide the stalling effects of long latency and blocking operations. Examples of long latency operations are loads from global or texture memories. Blocking operations include barrier synchronization, which stops a warp until all warps in the same block have reached the barrier.

A common optimization in this category is to decrease the thread block size and increase the number of thread blocks. This change can increase the number of thread blocks assigned to each SM and provide more independent warps from other blocks when one block is stalled. However, increasing the number of thread blocks can require changing the granularity of computation. In cases where memory tiling is utilized, it may have the effect of reducing data sharing and increasing pressure on memory bandwidth.

3.4. Increasing intra-thread parallelism

The fourth category of optimization, intra-thread parallelism, ensures the availability of independent instructions within a thread. These optimizations have a multiplicative effect with thread-level parallelism, so small changes can have major effects on performance. Intra-thread parallelism breaks down into two subcategories.

- *Instruction-level Parallelism (ILP).* ILP-increasing code transformations have been extensively studied and applied to many architectures, particularly EPIC/VLIW architectures. John Sias' Ph.D. dissertation [26] discusses many of these in detail. One interesting case is loop unrolling, which is also an efficiency-increasing optimization. An example is shown in Fig. 2(b), where the loads from shared memory for the different tiles can execute in any order.

  This subcategory is primarily the jurisdiction of the instruction scheduler of the CUDA runtime. It appears to reschedule operations to hide intra-thread stalls, but sometimes does this to the detriment of inter-thread parallelism. As with optimizations to reduce instruction count, scheduling to reduce intra-thread stalls may increase register usage and potentially reduce the number of thread blocks on each SM.
- *Memory Latency Hiding.* A special case of ILP, memory operations can be scheduled to overlap and amortize latencies, reducing the likelihood that execution resources will stall on global memory accesses. The GeForce 8800 supports a large number of in-flight memory accesses, making overlapping of memory access latencies a key aspect of performance on that architecture. This category breaks down into memory-level parallelism (MLP) [8] and prefetching.
  - Memory-level parallelism refers to the ordering of memory operations to ensure that many loads are in flight before a stalling use is encountered. For example, in Fig. 2(a), loads of elements of arrays `A` and `B` can be scheduled before the store to `As` in order to amortize the exposed latencies of the loads. This schedule also causes more work to be available from the same thread block during the stall, since each warp has two load operations prior to a consuming store. This optimization is generally the domain of nvcc and is difficult for a developer to control manually.
  - The developer can explicitly insert prefetching code to initiate long-latency global loads as early as possible and hide its latency with other instructions. An example is shown in Fig. 2(c). Prefetching generally requires additional register usage in code regions where register usage is already high, sometimes causing fewer thread blocks to reside on each SM.

3.5. Work redistribution

The fifth category involves redistribution of work across threads and thread blocks. These optimizations can change both efficiency and utilization and have some effect on memory bandwidth usage. Because of their nature, optimizations in this category can have unpredictable results due to changes in register usage.

Tiling and the choice of tile size is one optimization that also falls into this category. Although tiling is primarily done to improve memory bandwidth usage, the size of the tile also has efficiency and utilization implications. Assuming that each thread computes a single element of the output data set, the use of large memory tiles has higher efficiency than smaller tiles, but has reduced scheduling flexibility (utilization) since a larger percentage of threads must stall at barrier synchronizations.

Another example is rectangular tiling in Fig. 2(d): instead of each thread computing one result, they compute multiple results. This allows register tiling [7], where values can be reused in a register within a single thread, eliminating loads. The cost is reduced thread-level parallelism. Another benefit is the reduction in the total number of control flow operations: more work is done per block and the total number of blocks is reduced. There is often a small increase in ILP, but the increase in register usage often causes fewer thread blocks to be executed per SM.

Another optimization that is occasionally useful is to distribute work across multiple invocations of a kernel. This is sometimes necessary to use the constant cache, due to its size limitation. It may also help improve cache behavior. The additional kernel invocation overhead can be negligible.

3.6. Resource balancing

The last category is best termed resource balancing. The purpose of these optimizations is to make changes in resource usage to alleviate pressure on oversubscribed resources and shift to underutilized ones to produce better overall performance. Unless the whole system is taken into account, the optimization may be counterintuitive. Because of the high amount of execution resources, most optimizations involve decreasing instruction efficiency to optimize another aspect of execution and improve utilization.

One optimization in this category is the shifting of data from capacity-constrained, low-latency memories to larger, high-latency memories to allow more threads to run simultaneously. For example, proactive, explicit register spilling may be done by the application developer to allow more thread blocks to be assigned to each SM. The resulting configuration may achieve much higher performance, despite the reduced instruction efficiency, because the additional thread blocks improve overall resource utilization. Similarly, a kernel with small input and output data sets but large working sets may find it advantageous to spill data to global memory to avoid congesting registers or shared memory.

4. Optimization carving

When developing a program with significant performance requirements, a developer begins with a mental model of the target platform and creates the application with this in mind. One usually starts with the first-order concern for performance, which historically has been the application algorithm and its corresponding instruction count. After finding a seemingly reasonable base configuration, the application is tested and then iteratively optimized until an acceptable configuration is found.

This iterative approach is often adopted by compilers that generate high-performance code for traditional uniprocessors and multiprocessors: algorithms cannot be changed, so the efficiency of the instruction stream is the first-order performance concern. A plethora of instruction removal, redundancy elimination, and strength reduction optimizations have been developed to address this concern.

However, the iterative optimization approach has major issues when applied to GPU compilation. The broad issues were discussed in Section 1 and some specifics are discussed here:

1. It is well understood by the compiler community that particular orderings of optimizations can trap optimization processing into an application's local performance maximum. This has not received as much attention as it might have on a single superscalar or EPIC/VLIW processor, because the performance difference between two configurations might be on the order of several percent. On platforms such as the GeForce 8800, a performance difference of more than double is possible. Furthermore, optimizations for the system have complex effects, and a poor decision by an application developer or compiler can trap the application in a local performance maximum. This makes it imperative to perform a broad search of the space when increased performance is valuable.
2. GPUs have a wealth of execution resources, but these are separated both by physical space and by architecture capabilities capabilities, such as the lack of direct global-to-scratchpad memory transfers on the GeForce 8800. The cost of communication is significant, which is well-understood by parallel program optimization experts. However, the degree of tradeoffs is different: for example, there is a much more vague tradeoff between whether a processing unit should recompute a value locally or obtain that value from another processing unit, due to shorter communication latencies between processing units. Determining the best tradeoffs will be difficult for most developers.
3. The GPU innovation cycle is very short and new features are incorporated into products every several months. Some of these reside on hardware and remain disabled until their correctness can be validated. However, it takes time and effort for developers and compiler writers to utilize new features; they will not be well understood for a significant amount of time. Also, to preserve competitive advantage, limited information may be available about these features. During application development for the Geforce 8800, we were able to find little information on global memory coalescing, cache attributes, or SFU performance. Iterative compilation is likely to make the wrong assumptions about how to use these features and create suboptimal application configurations.

Instead of selecting a single starting configuration and iteratively optimizing it, we propose the use of a process we term *optimization carving*, described here.

4.1. Description of the technique

Conceptually, optimization carving begins with a large optimization space for an application. By examining metrics of application configurations for the system, it repeatedly removes configurations in the space that are unlikely to get good performance. Each carving prunes the optimization space, eventually leaving a few potentially high-performance configurations that can then be evaluated via hardware execution to determine the best one.

There are several reasons why optimization carving is practical for the GeForce 8800:

- The kernel codes studied here have a small number of independent configuration axes, resulting in a relatively small search space.
- The effects of code transformations, particularly in combination, are unpredictable because of the application developer's lack of control over the runtime's instruction scheduling and register allocation. Thus, iterative optimization is unlikely to find a configuration with close to the best performance.
- On single-chip, many-core architectures, configurations that are trapped in local performance maxima may be significantly removed from the optimal in both performance and transformations, as shown in previous work [23]. A partial search of the optimization space may provide substantial performance gains over iterative optimization.

Optimization carving is done in order of performance impact. First-order issues are addressed first, then second-order, and so on. We identify these issues in the following subsections. Carving must be done correctly to find a near-optimal configuration, but it is generally easier to understand the first- and second-order performance concerns of an architecture than the exact effects of optimizations. We begin with a complete optimization space and full knowledge of resource usage for this reason. How to perform more speculative optimization carving is left for future work.

There are two kinds of carving, *threshold carving* and *tradeoff carving*. Each has different selection criteria and is discussed below, along with how it manifests on the GeForce 8800.

4.1.1. Threshold carving

Threshold carving is performed when some performance aspect must be satisfied or mitigated in order to achieve good performance. The example on the GeForce 8800 is off-chip memory bandwidth: performance is not positively impacted by classical, efficiency-increasing optimizations when the hardware is continuously stalled on off-chip memory accesses.

Selection for threshold carving involves pruning all configurations that do not surpass either a relative or absolute threshold.

In the matrix multiplication example, the developer or compiler performing the carving could require that all loads must be coalescible, or that the estimated bandwidth requirement of the application be below a certain limit. This is an example of an absolute threshold. A relative threshold might be established for an application where memory bandwidth is always a bottleneck, in which case only the configurations with the most efficient memory usage are evaluated.

Global memory bandwidth usage on the GeForce 8800 is estimated by examining the percentage of memory accesses in the instruction stream and determining the average number of bytes being transferred per cycle in the absence of stalls. The global memory coalescing effect can be included by observing whether memory accesses in kernel code are to contiguous locations within a warp and are aligned (offset by a multiple of 16 from the beginning of the structure). For this work we prune configurations that do not perform global memory coalescing when configurations that enable coalescing exist.

4.1.2. Tradeoff carving

Tradeoff carving has a different nature from threshold carving: in some cases, it is not clear that one should maximize or minimize a particular performance aspect of an application on the architecture. One example is instruction stream efficiency: some redundant computation may improve performance by allowing more threads to execute in parallel or reducing communication between threads. The purpose of tradeoff carving is to balance two or more aspects of an application because the optimal balance is unclear.

Selection for tradeoff carving involves metrics for two or more aspects of an application. The configurations that are retained are those that lie on a Pareto-optimal curve: no point on the curve is inferior in every dimension to any other point in the space. Any configuration that is inferior in every dimension to any other configuration is pruned.

When the performance of the GeForce 8800 is not limited by memory bandwidth capacity, it is determined by two factors: instruction stream efficiency and execution resource utilization. In many cases one can be traded off for the other with potential improvement in performance. An example of this tradeof is the use of shared memory to capitalize on data reuse: instead of loading a value multiple times from global memory, a thread can store the value in shared memory (an extra instruction) after the first load and subsequently load the value from there. Even though an extra instruction must be executed to store a datum to the shared memory, the thread does not stall on global memory access for subsequent loads and, therefore, can make faster progress. On the other hand, if threads use too much shared memory, this configuration may cause fewer threads to be simultaneously executed per SM and thus reduce performance.

The concepts of efficiency and utilization are very general and can be applied to any computer architecture. For applications executing on the GeForce 8800, it is possible to calculate metrics that work reasonably well in practice. The versions developed for this work are explained below.

$$\text{Efficiency} = \frac{1}{\text{Instr} * \text{Threads}}. \tag{1}$$

Eq. (1) estimates the instruction efficiency of the kernel to be run on the GPU by counting the total number of instructions that will be executed. Instr is an estimate of the number of dynamic instructions that will be executed per thread on the GPU, derived from the PTX code generated by nvcc. For this work, the average iteration counts of the major loops in the kernel are annotated to obtain this data. Some instructions count as multiple instructions; 32-bit integer multiplication is the primary example, taking multiple processor cycles to execute on the GeForce 8800. Threads is the number of threads that will run on the GPU for a given problem size, known to the developer when writing the code. This is made explicit in the invocation of the kernel function and does not have to be an absolute value as long as the relative values of different configurations are correct.

In the absence of a memory bandwidth bottleneck and assuming nearly full SP utilization, we expect that efficiency will correlate directly to the performance of different configurations. Because it counts the total number of instructions executed, the metric measures the instructions that are redundant across threads and this penalizes configurations that have more redundancy, such as ones with finer-grained threads. This effect should be an accurate reflection of performance when the SPs are fully utilized.

$$\textit{Utilization} = \frac{\textit{Instr}}{\textit{Regions}} \left[\frac{W_{TB} - 1}{2} + (B_{SM} - 1)(W_{TB}) \right]. \tag{2}$$

Eq. (2) estimates the utilization of the compute resources on the GPU. The purpose of this metric is to indicate the schedulability of warps in the system. This is done by primarily looking at TLP and determining how often a warp is expected to wait and the amount of available work from independent warps. The fraction $\frac{\textit{Instr}}{\textit{Regions}}$ represents the average number of non-blocking instructions a single warp is expected to execute before encountering an instruction that causes it to stall. As before, *Instr* is the per-thread number of dynamic instructions that will be executed on the GPU. *Regions* is the number of dynamic instruction intervals delimited by blocking instructions or the start or end of the kernel. Examples of blocking instructions are those that consume the results of long latency operations (generally global and texture memory loads) and synchronization instructions. SFU instructions are considered to have long latency when longer latency operations are not present.

The quantity within the brackets represents the number of independent warps in the SM, other than the one currently executing, that can be executed while the blocking instruction is being resolved.

- The first term in the bracket is the number of other warps in the same thread block as the currently executing warp. W_{TB} is the number of warps in a thread block, which is determined by dividing the number of threads in a thread block by 32. There is a division by two because if the blocking instruction is a synchronization instruction, on average half of the warps in the same block still need to execute until they also reach the synchronization point. Although loads do not have this restriction, we assume worst-case behavior.
- The second term in the bracket is the number of warps in other thread blocks on the SM that can execute. B_{SM} is the number of thread blocks assigned to each SM. The runtime assigns the maximum number of thread blocks possible to each SM, up to eight, without violating local resource usage. This number can be calculated from the local resource usage obtained via `-cubin`.

Synchronization instructions are grouped with the consumers of long latency memory operations in order to simplify the calculation of the *Regions* term, even though they display different behavior. Execution at a barrier synchronization proceeds only when all of the threads in a thread block have reached that point, while global load operations execute immediately and do not block execution until a use of the destination operand is encountered. The division by two in the first term in the bracket is a worst-case assumption.

There is a distinct upper limit on how much the execution resources can be utilized. If utilization were used in a single cost metric (e.g., efficiency $*$ utilization), it would be expected that

the value would be capped or asymptotically approach the peak theoretical limit of 10 operations per cycle, per SM. However, because the intent is to use this metric as part of a Pareto-optimal selection, it is more important that the superiority or inferiority of a configuration relative to other configurations is retained. We do not cap the value because it is unnecessary for this purpose. Because of this decision, the relative utilization values of configurations may not be meaningful.

As discussed previously, running nvcc with `-cubin` and `-ptx` flags is faster than full compilation of an application. Computing the efficiency and utilization metrics is relatively fast after this information and a few numerical inputs from the developer or a profile are obtained. This allows for fast exploration of the search space.

The utilization metric simplifies some aspects of the architecture for which details are not available. It does not use instruction latency values or instruction schedules and makes very simple assumptions about the warp scheduling policy. During tradeoff carving, a good configuration may be undervalued because the metrics do not capture all performance aspects of the architecture. Using more detailed performance estimates, particularly those that introduce more variables into the calculations, *increases* the risk that any effects that are not exactly modeled will inflate the metric values for poor-performing configurations. This possibility of "false precision" means that metrics used for tradeoff carving should either be very broad or extremely correct. Another method for increasing the robustness of tradeoff carving is to retain configurations that are within a certain distance from the Pareto-optimal curve, although this increases the number of configurations to be evaluated. Exploration of the use of more precise metrics is described in Shane Ryoo's Ph.D. dissertation [24].

4.2. Applying metrics to matrix multiplication

The matrix multiplication kernel shown in Fig. 3 is used to demonstrate the calculation of the metrics. The kernel is first compiled with `-cubin` to obtain the resource usage, which shows that each thread uses 13 registers, and each block uses 2088 bytes of shared memory for its 256 threads. The number of blocks per SM is determined by referring to the per-SM resource limits in Table 2. In this case, register usage is the limiting factor: $B_{SM} = \lfloor 8192/(13 * 256) \rfloor = 2$. The number of warps per thread block is $W_{TB} = \lceil 256/32 \rceil = 8$.

This kernel is then compiled with `-ptx` to determine its execution profile. The loop is annotated with a trip count of 256, found by dividing the matrix size (4096) by the tile length (16). With this annotation, the number of dynamically executed instructions can be counted statically. A single thread runs 15,150 instructions, including 512 barriers and 256 load-consuming instructions, so *Instr* = 15,150 and *Regions* = 512+256+1 = 769. The final piece of information needed is the number of threads in the kernel. There is one thread for each element of the 4k-by-4k output matrix: *Threads* = 2^{24}. From these numbers, *Efficiency* = $3.93 * 10^{-12}$ and *Utilization* = 227. As previously stated, the relationship of these metrics among different configurations is more meaningful than their absolute values.

5. Experiments

This section presents the use of optimization carving to find high-performance configurations of several benchmarks. We discuss some of the shortcomings of the technique and illustrate them with a simple example. We also compare the technique to random sampling of the entire optimization space, and the use of different computations for the utilization metric.

```
Ctemp = 0;

// trip count = 256
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  // Region #1
  // It is assumed that both loads execute before
  // the stores, which mark the end of the region.
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];

  // Region #2
  indexA += 16;
  indexB += 16 * widthB;

  // This synchronization marks the end of region #2.
  __syncthreads();

  // Region #3
  Ctemp +=
     As[ty][0] * Bs[0][tx];
  ...
  Ctemp +=
     As[ty][15] * Bs[15][tx];

  __syncthreads();
  // This is the end of region #3.
}
C[indexC] = Ctemp;
```

Fig. 3. Matrix multiplication example for calculating metrics. This is the same as Fig. 2(b) with regions delineated for clarity.

5.1. Methodology

The results in this section were obtained with CUDA version 1.0. Experiments were performed on an Intel Core2 Extreme Quad running at 2.66 GHz with 4 GB of main memory. The presented data represent runs with smaller inputs than those considered typical, which allowed us to explore the entire optimization space in a reasonable amount of time and determine the proximity of our selected configurations to the highest-performing one. Informal experiments have shown that execution time will scale accordingly with an increase in input data size for these applications on this architecture, due to the regular and otherwise data-independent execution of the kernels. The data are gathered from a single run of each configuration; repeated runs have shown that the gathered runtimes are reliable.

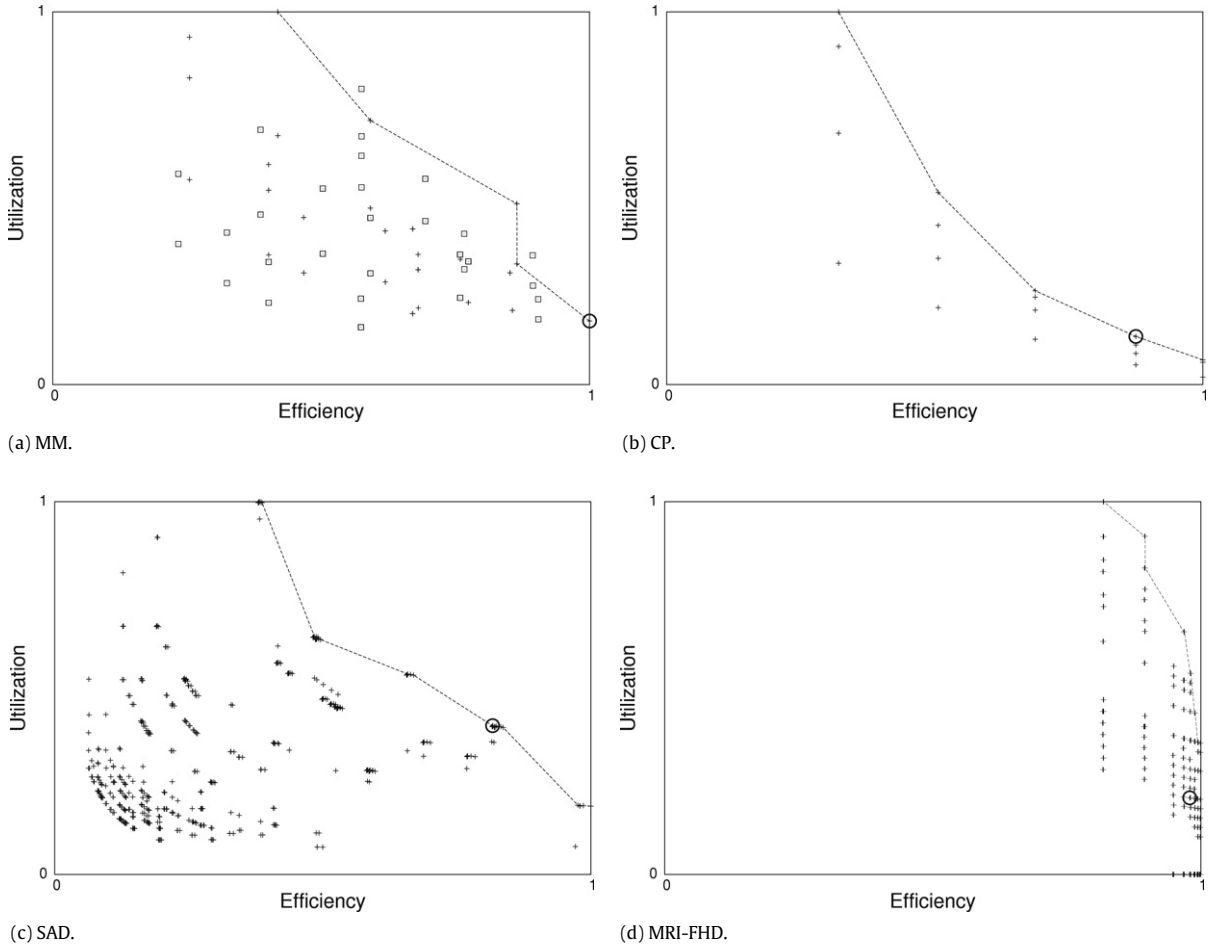
Benchmark applications were selected from those presented in previous work [25]. Table 3 lists the applications, optimization parameters varied (ones with no effect were excluded), the number of configurations in the optimization space, and the total time needed to evaluate the performance of every configuration in the optimization space. These applications were selected for their large number and variety of applicable optimizations, although none of the search spaces presented here are provably complete. Optimizations were performed only on the kernel source code. References to the "best" configuration indicate the configuration with the highest performance in the optimization space.

5.2. Results

Fig. 4 shows plots of the metric values for each optimization configuration for all of the applications. The maximum metric value along each axis has been normalized to one for comparison purposes. In general the best performance should come from configurations with both high efficiency and utilization, meaning points towards the right and/or the top of the graph. The configuration with the best performance for each application is circled in Fig. 4 for higher visibility. The average values of efficiency and utilization are different for each benchmark, reflecting the difficulty of establishing a simple cost function to find the best configuration.

Table 3
Applications and parameter search properties

Kernel	Parameters varied	Total configurations	Total evaluation time (s)
Matrix Multiplication (MM)	Thread block/memory tile size, rectangular tile dimension, unroll factor, prefetching, register spilling	93	363.3
Coulombic Potential grid (CP)	Thread block size, per-thread tiling, coalescing of output	38	159.5
Sum of Absolute Difference (SAD)	Per-thread tiling, unroll factor (3 loops), work per thread block	908	7.677
Magnetic Resonance Imaging reconstruction (MRI-FHD)	Thread block size, unroll factor, work per kernel invocation	896	2875



(a) MM. (b) CP.

(c) SAD. (d) MRI-FHD.

Fig. 4. Optimization carving for four Benchmark applications. The best performing configuration is circled in each graph. Configurations pruned by threshold carving are marked with squares rather than '+'. In (b), each point represents as many as seven configurations with indistinguishable efficiency and utilization.

The matrix multiplication kernel has thread blocks of size 8×8 and 16×16 in its configuration space. Although a developer with detailed knowledge of global memory coalescing would know to exclude 8×8 blocks, we were not aware of the specific constraints of coalescing when these experiments were first performed. We use this opportunity to show the effects of threshold carving: configurations pruned by threshold carving in MM are marked with an square rather than '+'. These configurations run significantly more slowly than the plot would indicate because they are limited by memory bandwidth. A Pareto-optimal curve that includes these configurations will still find the best configuration, but at the cost of evaluating several configurations with poor performance. The other kernels had naturally coalesced access patterns and were not affected by threshhold carving.

Table 4 shows the number of configurations selected by carving and the resulting reductions in space and evaluation time. The reductions were significant. The table also shows the performance of the best configuration in the Pareto-optimal curve compared to the best performance of all configurations, found via exhaustive search. For three applications, the Pareto-optimal subset contains the best configuration. The configuration with the best performance in the MRI-FHD kernel does not lie on the Pareto-optimal curve, but the second-best configuration does, with a performance difference of less than 1%. The variation in runs is relatively close to this difference, and there are several points on the Pareto-optimal curve that are within 2% of the best performance.

It is difficult to make a judgment about whether a given value of utilization is "good" or "bad". As previously stated, the utilization metric measures the relative ability of a configuration to utilize the execution units, but the differences in value are of less importance. For example, consider that the best configuration for matrix multiplication in Fig. 4(a) has one of the lowest utilization values of the entire space. The reason low utilization is not a reliable predictor of poor performance is because all configurations

Table 4
Optimization carving space reduction

Kernel	Selected configurations	Space reduction (%)	Selected evaluation time	Time reduction (%)	Selected best relative to overall best (%)
MM	8	91	10.2 s	97	100
CP	10	74	42.95 s	73	100
SAD	19	99	62.21 ms	99	100
MRI-FHD	58	93	270.0 s	91	99.2

Table 5
Random sampling results

Kernel	Best configuration	Optimization carving configs	Expected maximum performance of same size random sample	Performance relative to best (%)	Random sample size for 90% of best	Random sample size for 95% of best
MM (16×16 only)	1.147 s	8	1.404 s	81.7	20	37
CP	2.679 s	10	2.869 s	93.4	8	15
SAD	2.029 ms	19	2.479 ms	81.8	78	93
MRI-FHD	3.727 s	58	3.763 s	99.0	2	4

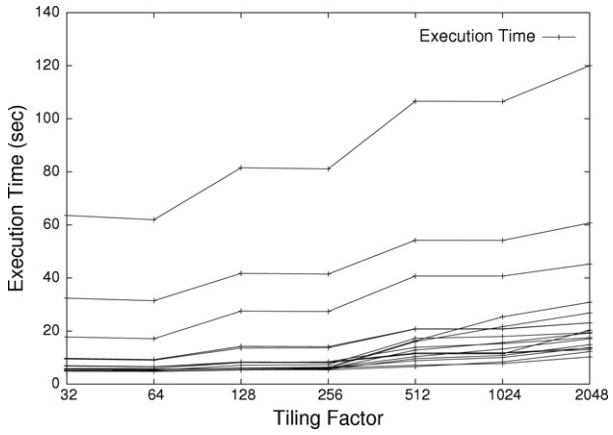


Fig. 5. Execution time of MRI-FHD with cache conflicts.

are quite good at ensuring that execution resources are almost always occupied. Thus, efficiency is the primary determinant of performance for this benchmark. This is not necessarily true for the other applications.

Fig. 4(d) shows the metric plot for the MRI-FHD application. In this graph, configurations tend to be clustered in groups of seven because changing the tiling factor affects neither the efficiency nor the utilization of this benchmark, appearing as a single point at this resolution. Differences in actual performance within each cluster are small, with the maximum variation in performance among configurations within a cluster being 7.1%. Hence, when several configurations have identical or nearly identical metrics and similar optimization parameters, it may be sufficient to randomly select a single configuration from that cluster, rather than evaluating all the configurations.

The utilization metric does not reflect optimizations' effect on instruction latency. The kernels used to evaluate optimization carving have been written to avoid cache conflicts and stalls caused by contention for hardware resources. However, in some cases these hardware interactions may cause instruction latency to vary systematically across the optimization space. Consider Fig. 5, which depicts the performance of a preliminary version of MRI-FHD as the tiling factor (number of data points processed by each thread block) changes. The performance metrics indicate that efficiency and utilization remain constant as the tiling factor changes, predicting no significant change in performance. However, Fig. 5 shows that performance worsens dramatically as the tiling factor increases.

The sharp contrast between the predicted performance trend and the actual performance led us to consider that the layout of data in constant memory might be causing frequent constant cache conflicts. Configurations with a smaller tiling factor had smaller working sets and fewer cache conflicts, and consequently a faster effective constant memory access latency. Changing the data layout yielded a kernel that is insensitive to changes in the tiling factor and 17% faster than the previous best configuration. Incorporating cache effects into optimization carving will be addressed in future work.

5.3. Comparison to random sampling

The last stage of optimization carving requires execution of the remaining configurations to determine the best-performing one. A question that arises is whether a random sample of the optimization space might be capable of achieving similar results. In this section we compare random sampling of the optimization space to the results of optimization carving.

The value of interest in random sampling is the highest performance of the configurations in the sample, or the *expected maximum performance*. Fig. 6 shows the expected maximum performance of random samples of varying size, while Table 5 shows the number of samples required to have an expected maximum performance of 90% and 95% of the best configuration. For matrix multiplication we used only 16×16 thread blocks, since thread blocks that do not take advantage of global memory coalescing have mediocre performance and can be eliminated prior to sampling. A vertical dotted line corresponds to the number of selected configurations from optimization carving, shown for comparison.

We find that random sampling is an effective method for finding good configurations for both CP and MRI-FHD, with an expected maximum performance close to the best configuration's performance with a small random sample. This is because there are many configurations with near-best performance in the optimization spaces of those benchmarks.

Matrix multiplication and SAD require much larger random samples before a near-best configuration can be expected. Unlike CP and MRI-FHD, there are only a few configurations with performance near the best, and even these can still be several percent inferior. The right combination of local memory tiling, register tiling, and loop unrolling gives significant performance advantages to the highest-performing configurations.

6. Related work

Code transformation and optimization for parallel programs have a long history, with much of the foundational work performed by the Parafrase [18], PTRAN [2], and PFC [4] projects. Many