Singe: Leveraging Warp Specialization for High Performance on GPUs

Michael Bauer
Stanford University
mebauer@cs.stanford.edu

Sean Treichler
Stanford University
sjt@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

We present Singe, a Domain Specific Language (DSL) compiler for combustion chemistry that leverages *warp specialization* to produce high performance code for GPUs. Instead of relying on traditional GPU programming models that emphasize data-parallel computations, warp specialization allows compilers like Singe to partition computations into sub-computations which are then assigned to different warps within a thread block. Fine-grain synchronization between warps is performed efficiently in hardware using producer-consumer named barriers. Partitioning computations using warp specialization allows Singe to deal efficiently with the irregularity in both data access patterns and computation. Furthermore, warp-specialized partitioning of computations allows Singe to fit extremely large working sets into on-chip memories. Finally, we describe the architecture and general compilation techniques necessary for constructing a warp-specializing compiler. We show that the warp-specialized code emitted by Singe is up to 3.75X faster than previously optimized data-parallel GPU kernels.

Categories D.1.3 [*Programming Techniques*]: Parallel Programming
Keywords warp specialization; warp-specializing compiler; GPU; DSL

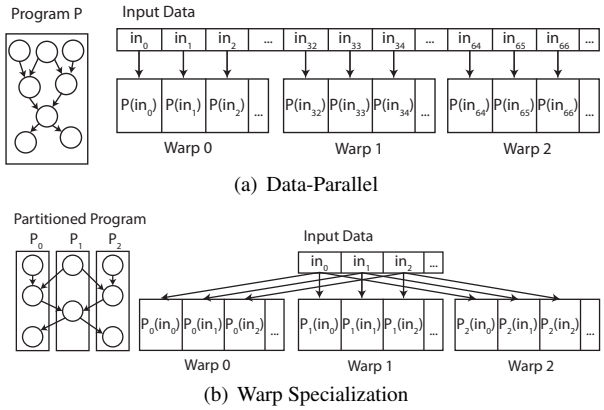
1. Introduction

Current GPU programming models, such as OpenCL[10] and CUDA[1], support data-parallel computations where all threads execute the same instruction stream on arrays of data. However, the expansion of GPUs into general purpose computing has uncovered many applications which exhibit properties which make them challenging to map onto traditional data-parallel GPU programming models. For example, consider the domain of combustion science, which includes applications such as S3D[5, 11]. The physics and chemistry kernels for these combustion applications have three characteristics that make them difficult to optimize for GPUs:

- Large working sets: combustion kernels routinely require hundreds of live double-precision variables per physical point in discretized space. In a data-parallel model these working sets commonly exceed the small on-chip memory capacity allotted to each thread, resulting in register spilling, low occupancy, and under-utilization of math units.

PPoPP '14, Feburary 15-19, 2014, Orlando, FL, USA.

ACM 978-1-4503-2656-8/14/02. . . $15.00.
http://dx.doi.org/10.1145/2555243.2555258



(a) Data-Parallel

(b) Warp Specialization

Figure 1. Contrasting GPU programming models.

- Irregular computations: combustion kernels often contain multiple phases with different computational characteristics. The large number of temporaries between phases necessitates that the phases be fused together into a single kernel. While some of these phases could be run in parallel, current data-parallel GPU programming models serialize these phases.
- Irregular data accesses: data accesses in combustion kernels are dependent both on the properties of the chemical mechanism as well as runtime values. Under many circumstances it is impossible for a data-parallel model to avoid memory divergence and shared memory bank conflicts.

In practice, many different scientific computing domains demonstrate some or all of the same characteristics as combustion. Achieving peak performance for these applications mandates finding alternative approaches to programming GPUs.

In this work we describe how *warp specialization* can be used as an alternative programming model for mapping irregular and large working set applications onto GPUs. Figure 1 illustrates the differences between the core data-parallel abstraction of current GPU programming models and warp specialization. In the data-parallel model, a collection of threads within a *thread block* all execute the same program over independent elements from arrays of input data. On the hardware, however, a thread block is broken into *warps* consisting of (typically) 32 threads which serve as the unit of scheduling. Warp specialization exploits the division of a thread block into warps to partition computations into sub-computations such that each sub-computation is executed by a different warp within a thread block. Carefully structured programs can handle irregularity by grouping threads into warps such that threads within a warp have good data-parallel behavior, even if threads in different warps do not. Warp specialization can also be used to partition extremely large working sets across multiple

threads in different warps, keeping data on-chip and dramatically reducing the register pressure on an individual thread.

We describe the design and implementation of a Domain Specific Language (DSL) compiler capable of using warp specialization in conjunction with domain specific knowledge to better map challenging kernels onto GPU architectures. Our approach has been implemented in Singe, a DSL compiler for general combustion simulations. While Singe targets combustion specifically, the techniques described in this paper are general and can be adapted to other domains. We show that using these techniques, Singe generates warp-specialized code that achieves speedups up to 3.75X over heavily optimized but purely data-parallel combustion kernels.

In Section 2 we give a brief introduction to the mechanics of warp specialization. Each of the following sections describe one of our primary contributions.

- We present a case study of how Singe uses warp specialization to partition the three most expensive kernels in a real-world combustion application. We elucidate how warp-specialized partitioning addresses the critical performance bottlenecks in these kernels (Section 3).
- We describe the architecture of a warp-specializing compiler including the necessary algorithms for managing data placement, communication, and synchronization for general warp-specialized kernels (Section 4).
- We cover three general techniques for generating high performance warp-specialized code that are essential to avoiding instruction cache thrashing. We give code examples of how Singe employs these techniques (Section 5).
- We investigate the performance advantages conferred by warp specialization by examining the performance of the kernels emitted by Singe on both Fermi and Kepler GPUs for two different chemical mechanisms (Section 6).

Section 7 discusses implications of warp specialization, Section 8 describes related work, and Section 9 concludes.

2. Warp Specialization

While there are several APIs for programming GPUs, they all implement variations of the same programming model. We use CUDA as a proxy for the standard GPU programming model as it is the only interface that currently supports the fine-grain synchronization primitives necessary for warp specialization.

CUDA launches grids of thread blocks or *cooperative thread arrays* (referred to as CTAs for the remainder of the paper) on the GPU. This abstraction gives the hardware considerable flexibility when executing a CUDA application. In current GPUs, the threads within a CTA are broken into groups of 32 threads called *warps*. All threads within a CTA (and therefore also within a warp) execute the same program. If the threads within a warp *diverge* on a branch instruction, the streaming multiprocessor (SM) on which the warp is executing first executes the warp with all the threads not taking the branch masked off. After the taken branch is handled, the warp is re-executed for the not-taken branch with the complementary set of threads masked off from executing. Divergence is severely detrimental to program performance because it serializes potentially parallel thread execution within a warp.

The crucial insight for warp specialization is that while control divergence within a warp results in performance degradation, divergence between warps does not. A warp-specialized kernel is one in which dynamic branches, dependent on each thread's warp ID, create explicit inter-warp control divergence for the purpose of executing different code in each warp. As long as all threads within a warp execute the same instruction stream then the only execution overhead is the cost of the warp-specific branch instructions.

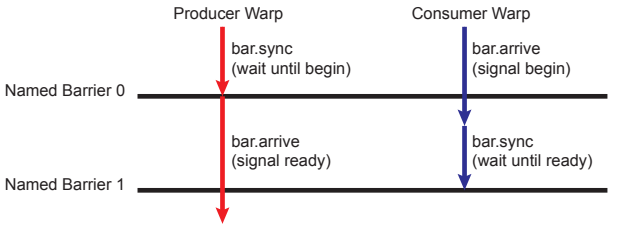


Figure 2. Producer-consumer named barriers example.

Warp-specialized programs also require more expressive synchronization mechanisms. In traditional CUDA programs synchronization is performed using barriers between all the threads within a CTA. However, by using inline PTX statements, a CUDA program has access to a more expressive set of intra-CTA synchronization primitives referred to as *named barriers*[2]. Named barriers provide two operations: *arrive* and *sync*. Arrive is a non-blocking operation which registers that a warp has arrived at a barrier and then continues execution. Sync is a blocking operation that waits until all the necessary warps have arrived or synced on the barrier.

Using arrive and sync operations, programmers can encode producer-consumer relationships in warp-specialized programs. Figure 2 illustrates using two named barriers to coordinate movement of data from a producer warp (red) to a consumer warp (blue) through a buffer in shared memory. The producer warp first waits for a signal from the consumer warp that the buffer is empty. The consumer warp signals the buffer is ready by performing a non-blocking arrive operation. Since the arrive is non-blocking, the consumer warp is free to perform additional work while waiting for the buffer to be filled. At some point the consumer warp blocks on the second named barrier waiting for the buffer to be full. The producer warp signals when the buffer is full using a non-blocking arrive operation on the second named barrier. It is important to note that named barriers support synchronization between arbitrary subsets of warps within a CTA, including allowing synchronization between a single pair of warps as in this example.

3. Warp-Specialized Partitioning

Warp-specialized partitioning provides a useful mechanism for DSL compilers when grappling with computations that exhibit both irregularity and large working sets. While warp-specialized partitioning is a useful tool, it is important to note that the particular method for partitioning a computation into specialized warps relies on both domain specific knowledge and the target architecture. Therefore the partitioning strategy must vary with each DSL compiler and we cannot provide a general partitioning algorithm. Instead, we provide a case study of coupling domain specific information with warp specialization to address performance problems in the domain of large and complex combustion applications. Subsequent sections will cover generally applicable techniques for mapping and scheduling computations after they have been partitioned using domain specific knowledge.

We begin with a brief overview of the combustion domain in Section 3.1. We then describe how warp-specialized partitioning is used by Singe to address the performance challenges inherent in the generation of three expensive combustion kernels: viscosity (Section 3.2), diffusion (Section 3.3) and chemistry (Section 3.4).

3.1 Combustion Chemistry

Combustion simulations are described by *chemical mechanisms* which consist of a set of *reactions* and the *species* involved in those reactions. Chemical species range from single elements to very large and complex hydro-carbons. Table 3 shows the characteristics of the Dimethyl Ether (DME) and n-Heptane mechanisms used in

Mechanism	Reactions	Species	QSSA	Stiff
DME	175	39	9	22
Heptane	283	68	16	27

Figure 3. Chemical Mechanisms

this paper. These mechanisms were chosen for their relevance in current combustion research[12].

Combustion mechanisms are described by a declarative data DSL based on the CHEMKIN[9] standard. A mechanism in this DSL is described by three input files:

- CHEMKIN file: a list of chemical reactions with stoichiometric coefficients and reaction models (see Figure 4)
- TRANSPORT file: a table of diffusion and viscosity coefficients for all chemical species
- THERMO file: a table of thermodynamic coefficients for all chemical species

Singe parses these files and emits CUDA code for each of the kernels necessary to simulate combustion of the specified chemical mechanism. Singe may also take an optional fourth input file describing the set of *quasi-steady-state approximation* (QSSA) and *stiffness* (Stiff) species. QSSA species arise out of techniques that reduce the total number of species across all phases of the simulation at the expense of additional computation during the chemistry phase of the application[12]. For example, in the heptane mechanism, the 16 QSSA species are removed from the original group of 68 so that only a total of 52 species must be simulated, while requiring a complex QSSA computation be performed in the chemistry kernel. Stiffness species allow the simulation to take longer time steps, but require additional computations be performed in the chemistry kernel. We describe how warp specialization handles both the QSSA and stiffness computations in Section 3.4.

Most combustion simulations operate on a three dimensional cartesian grid. Each point in the grid has an associated set of *fields* with each field laid out contiguously in a separate array to ensure coalescing of global memory loads. In most GPU combustion kernels[11], each thread is responsible for a single point in the cartesian grid, which conforms to the traditional data-parallel GPU programming model.

3.2 Viscosity

The viscosity kernel computes a viscosity coefficient for each point in the cartesian grid as a function of the temperature and molar fractions for each species. Per-species viscosities are first computed by taking the exponent of a per-species third order polynomial dependent on temperature (T):

$$vis_i(T) = e^{\eta_{i0}+\eta_{i1}*T+\eta_{i2}*T^2+\eta_{i3}*T^3}$$

The final viscosity output ν for each cartesian grid point is then computed as an interaction of the viscosity of each species with every other species given by the following equation:

$$\nu = \sqrt{8} * \sum_{k=1}^{N} \left[\frac{x_k * vis_k}{\sum_{j=1}^{N} x_j * \frac{\left(1+\sqrt{\frac{vis_k}{vis_j}} * \sqrt{\frac{m_j}{m_k}}\right)^2}{\sqrt{1+\frac{m_k}{m_j}}}} \right]$$

where N is the number of species and x_i and m_i are the molar fraction and molecular mass of species i respectively. In an optimized CUDA implementation, this computation is performed in logarithmic space to reduce the dependency on square root and divide operations that are implemented using Newton's method in the absence of dedicated hardware on GPUs. After constant folding, for each of the N^2 pairs of species the viscosity kernel requires that 2

```
!1
  ch3+h(+m) = ch4(+m) 2.138e+15  -0.40  0.000E+00
            low / 3.310E+30 -4.00 2108. /
     troe/0.0  1.E-15  1.E-15  40./ h2/2/ h2o/5/
!2
  ch4+h  =  ch3+h2  1.727E+04  3.00  8.224E+03
    rev /  6.610E+02  3.00  7.744E+03 /
!3
  ch4+oh  =  ch3+h2o  1.930E+05  2.40  2.106E+03
    rev /  3.199E+04  2.40  1.678E+04 /
...
```

Figure 4. Example CHEMKIN input file to Singe.

double precision constants be loaded and that 2 double precision adds, 2 double precision multiplies, and 10 double precision fused multiply-add (DFMA) operations be performed.

The viscosity computation is embarrassingly parallel as each point in the grid can be computed independently, but it places a significant strain on several GPU resources. First, the working set for a cartesian grid point is difficult to fit on chip into a single thread's registers. For the heptane mechanism with 52 species, storing the molar fractions and per-species viscosity values requires 104 double precision values, which would require 208 registers on an NVIDIA GPU. Fermi GPUs only support 64 registers per thread, while Kepler GPUs support 256 but at the cost of extremely low warp occupancy and under-utilized math units. In this scenario, a data parallel GPU programming model forces the DSL compiler to choose between low-occupancy or spilling registers, which adds additional memory latency to the kernel.

A second problem encountered by the viscosity kernel has to do with the large number of constant values required for the computation. Every pair of species requires two different double precision constants. GPU architectures include constant caches, but their working set sizes are small. GPUs only have 8 KB of on-chip constant cache[1]. The DME and Heptane mechanisms require 13.9 and 42.4 KB of constants respectively. Loading constants for the viscosity kernel is therefore expensive since they are unlikely to hit in the constant cache. The problem of hiding these long-latency constant loads is exacerbated by the low occupancy of the kernel caused by the large working set described earlier.

Singe uses warp-specialized partitioning to solve both of these problems. The outer sum over the set of chemical species is broken into individual computations each of which is mapped to a different warp using the algorithm described in Section 4.1. Unlike data-parallel CUDA where each thread handles a single point, all of the warps in a CTA cooperate on a set of 32 points. A thread in lane l of warp w handles the per-species computations assigned to warp w for the l-th point. Warp specialization necessitates the sharing of data between warps, therefore the molar fractions and per-species viscosities are moved into shared memory. These values fit easily because each CTA is only handling 32 grid points at a time.

Partitioning the computation for warp specialization also provides a solution to the problem of storing the large number of constant values in on-chip memory. Since each warp is only performing a subset of the computation, it only requires a subset of the constant values. Furthermore, moving the working set to shared memory makes the registers available for storing constants. Using the constant deduplication optimization described in Section 5.2, warp specialization enables all the constant values to be stored in on-chip registers. At the end of the computation, all the warps reduce their values through shared memory and the threads in warp 0 perform the write of the resulting values for each point. Using warp specialization, Singe is able to partition the viscosity computation so that values better fit into the on-chip memories which we will show leads to significant performance gains in Section 6.

X	X	X	2
1	X	X	X
1	1	X	X
X	1	2	X

X	X	X	2	2
1	X	X	X	2
1	1	X	X	X
X	1	1	X	X
X	X	1	2	X

Figure 5. Diffusion partitioning between two warps for N=4,5.

3.3 Diffusion

The diffusion computation shares some characteristics with the viscosity computation which leads to similar challenges, but requires a more complex warp-specialized partitioning scheme. For every pair of species i and j, a diffusive constant is computed from the exponent of a third order polynomial dependent on the temperature T where δ is a NxNx4 matrix of coefficients.

$$d_{ij}(T) = e^{\delta_{ij0}+\delta_{ij1}*T+\delta_{ij2}*T^2+\delta_{ij3}*T^3}$$

The NxN matrix of these diffusive constants is then used in computing the per-species diffusion outputs Δ_i:

$$mass = \sum_{j=1}^{N} m_j * x_j$$

$$clamp_i = max(\epsilon, x_i)$$

$$\Delta_i = \frac{P_{atmos}}{P} * \frac{-clamp_i * m_i + \sum_{j=1}^{N} clamp_j * m_j}{mass * \sum_{j=1}^{N} clamp_j * d_{ij}}$$

where N is the number of species, P is the pressure, P_{atmos} is atmospheric pressure, ϵ is the minimum molar fraction, and x_i and m_i are the molar fraction and molecular mass of species i respectively. Note that unlike the viscosity computation, which computed a single output value per point, the diffusion kernel computes one output value per species per point. Furthermore the d_{ij} matrix is symmetric with zeros along the diagonal which implies that less than half of the matrix must actually be computed. However, each d_{ij} must contribute to both Δ_i and Δ_j.

Like viscosity, diffusion suffers from the same large working set and constant problems. We again solve them using warp specialization, but with a different partitioning strategy. The NxN matrix of d_{ij} values is partitioned by column. Columns are offset by one from each other and only a subset of values in each column need be computed because of the symmetric nature of the matrix. For odd numbers of species, each column must compute $\lfloor \frac{N}{2} \rfloor$ d_{ij} values; for even numbers of species the first $\frac{N}{2}$ columns compute $\frac{N}{2}$ d_{ij} values and the second $\frac{N}{2}$ columns compute $\frac{N}{2} - 1$ d_{ij} values. Figure 5 shows the assignment of d_{ij} value computations for matrices with $N = 4$ and $N = 5$ for two warps. 'X' values indicate that the point need not be computed because its symmetric point about the diagonal has already been computed.

Warps are assigned adjacent columns to maximize locality. Each warp traverses its set of columns and maintains a partial sum for the species in each of its columns. Additional partial sums for each species are also maintained in shared memory. For a given row, the warp computes the d_{ij} values and reduces them into the partial sums stored in registers for each column. The warp also computes a row partial sum and reduces it into the location for the corresponding row species in shared memory. To avoid data races when accessing shared memory, named barriers are used to synchronize access to different regions of shared memory.

At the end of the computation, each warp reads the partial sums out of shared memory for the species that it owns and sums the results with the partial sums stored in the warp's registers. The pressure scaling ratio is then applied and each warp writes out results for its owned species. In this scenario, the warp-specialized partitioning of the diffusion computation results in a hybrid storage of the working set using both shared memory and registers that enables additional parallelism to be extracted. We describe the algorithm for placing data in further detail in Section 4.1.

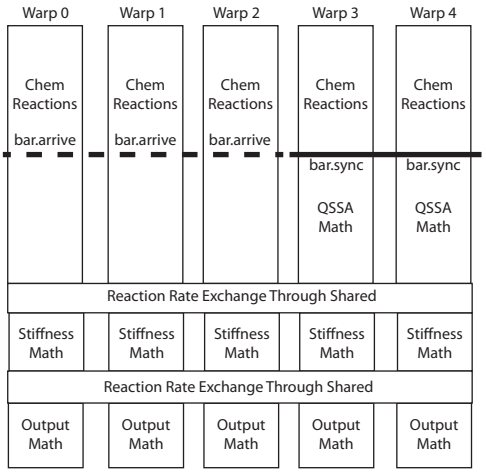


Figure 6. Warp specialization for chemistry kernel.

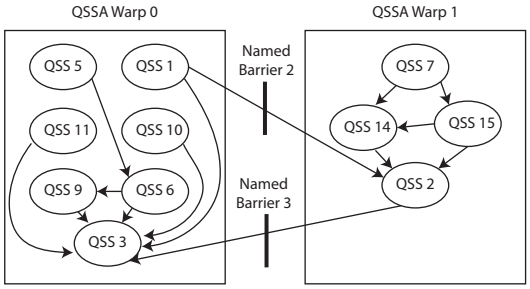


Figure 7. Example partitioning of heptane QSSA computation.

3.4 Chemistry

The chemistry computation is the most complex kernel that Singe emits because it requires multiple phases, some of which can execute in parallel. The first phase computes forward and reverse reaction rates for every reaction using an Arrhenius, Lindermann, or Landau-Teller reaction model[9], requiring between 6 and 15 double precision constants per reaction. The second phase computes the QSSA scaling factors and then applies them to all reactions involving a QSSA species. A similar computation is performed in the third phase for the stiff species. The output phase sums the contributions from each reaction and computes the resulting rate of change for each species based on stoichiometric coefficients.

While the chemistry kernel could be fissioned into separate kernels, each such kernel would read and write a forward and reverse reaction rate for every reaction at every point. Kernels for the heptane mechanism would each require 566 double precision reads and 566 double precision writes per cartesian grid point, which would be memory bandwidth limited and therefore slow.

Instead we employ warp-specialized partitioning to break apart the computation and the working set so it fits in on-chip memory. However, unlike viscosity and diffusion, for all interesting mechanisms the number of reaction rates is too large to store in shared memory. Instead we partition the reaction computations across warps and store the resulting forward and reverse reaction rates in each warp's registers. When reaction rates are needed for computa-

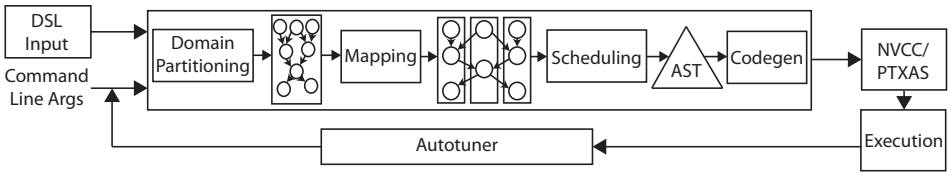


Figure 8. Architecture of a warp-specializing compiler.

tions, they are exchanged in passes by writing subsets of the rates into shared memory and having the needed values read out before progressing to the next pass. While this adds latency to the kernel, it is significantly less latency than going to off-chip global memory and well within the bandwidth limits of shared memory.

The chemistry kernel also suffers from another performance challenge: the QSSA phase is both computationally expensive and difficult to parallelize. Singe uses warp specialization at two different granularities to address this problem. First, the QSSA computation usually requires between half and two-thirds of the reaction rates. Singe assigns these reactions to warps first. A subset of the warps are then siphoned off to perform the QSSA computation while the remaining warps complete the reaction computations. Figure 6 shows an example of this partitioning of the warps for the chemistry kernel. A producer-consumer named barrier is used to indicate to the QSSA warps when the needed values from the non-QSSA warps have been written into shared memory. Note that because this barrier is non-blocking warp specialization enables the QSSA computation to be overlapped with the remaining reaction rate computations, which cannot be done in the data-parallel version of the kernel.

Warp specialization is also used to further parallelize the QSSA computation. The QSSA computation performs many divide operations and consequently requires between 20 and 60 DFMA operations for each QSSA species. Data dependences exist between species further complicating parallelization. Singe uses a heuristic for partitioning the directed acyclic graph (DAG) of a QSSA computation across a set of QSSA warps. For every edge that crosses a warp boundary, a producer-consumer named barrier is allocated. Figure 7 shows an example of one such partitioning for the QSSA computation for the heptane mechanism across two warps. In Section 4.2 we describe a general algorithm for synchronizing and scheduling these dataflow DAGs that avoids deadlock.

4. Warp-Specializing Compiler Architecture

We now describe the general architecture and compilation stages of a warp-specializing DSL compiler as seen in Figure 8. We are primarily interested in the transformations necessary for performing warp specialization and therefore assume a source-to-source compiler that will rely on a lower-level compiler like the PTX assembler to perform optimizations on sequential code within a single thread. Input for a warp-specializing compiler consists of a DSL file describing the computation to be performed and a set of command line flags indicating the number of warps to target as well as any explicit mapping decisions (described in Section 4.1).

Our experience with Singe has shown it is valuable for a warp-specializing compiler to generate correct code for any number of warps and choice of mapping decisions. This property enables autotuning and significantly reduces the complexity of the compiler by removing specialized logic for trying to compute an optimal mapping of a computation onto an arbitrary architecture. In practice, the search space for Singe was never more than a few hundred points because warp-specialized decisions dealt with very coarse-grained properties such as the number of target warps. Consequently, we used a brute-force exhaustive autotuning script to drive Singe when tuning our kernels.

The first stage of any warp-specializing compiler partitions the primary computation into sub-computations. As we have seen with the three kernels discussed in Section 3, how to partition and at what granularity is domain specific and will therefore be determined by each DSL compiler individually. The first stage outputs a dataflow graph with nodes corresponding to units of computation, which we refer to as *operations*, and edges indicating data dependences between operations. Section 4.1 describes the second stage of compilation which maps operations onto warps and determines where data is stored. The output of the second stage is another dataflow graph with each operation assigned to a specific warp and inputs and outputs of operations assigned either to registers or shared memory. The third compiler stage described in Section 4.2 performs named barrier placement and scheduling. The result of the third stage is an abstract syntax tree (AST) for each warp which encodes the chosen schedule and necessary synchronization for each warp's operations. Code is generated directly from these ASTs in the last compilation stage, but requires several non-standard traversal techniques which we cover in detail in Section 5.

4.1 Computation and Data Mapping

The mapping compilation stage is responsible for taking in an arbitrary dataflow graph of operations and mapping it onto the specified number of warps and available GPU memories. This is accomplished in two steps: first, operations must be assigned to warps, and second, the data values produced and consumed by operations must be assigned to one of the available software-managed on-chip GPU memories (e.g. registers, shared).

When performing the first step there are three primary (and often conflicting) metrics to consider when assigning operations to warps. First, the mapping stage should aim to achieve a balanced computational load across all the warps as imbalance can lead to under-utilized computational resources. In Singe, we use the number of floating point operations (FLOPS) in each operation as a proxy for computational load, and attempt to balance the total number of FLOPS assigned to each warp.

The second metric which must be balanced is the total registers required for each warp. The need for this metric is an artifact of the use of data-parallel programming models: current GPU architectures and compilers do not support per-warp register allocation schemas. Consequently, the warp requiring the most registers dictates the number of registers allocated for all warps in the kernel. If there is a large imbalance in required registers between warps, then significant fractions of the register file can go unused, limiting the total working set size for warp-specialized kernels and reducing total occupancy. In Singe, we assume intermediate values in an operation consume no registers because they are short-lived, but the values generated by operations consume registers as long as they are live. Singe stores values generated by an operation in the registers of the thread that computed the operation to avoid duplicating values and consuming additional registers.

The third metric which must be considered is locality. A warp-specializing compiler should attempt to maximize locality by minimizing the number of dataflow edges with operations assigned to different warps. This reduces the amount of communication that must occur through shared memory and also reduces the number of barrier synchronizations which must be performed.

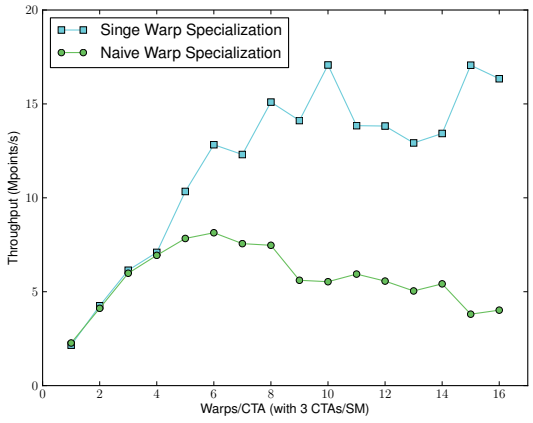


Figure 9. Comparison of warp-specialized code generation.

There are many possible algorithms for mapping operations onto warps that consider all three metrics, including some which may opt to use domain- or architecture-specific knowledge. The Singe compiler uses a general greedy algorithm for performing mapping. Operations are weighted based on their FLOP counts, register needs, and locality to warps based on the current mapping. Singe maps operations in order of cost from the most expensive to the least in a way that locally minimizes overall cost. Singe makes mapping visible to the autotuner by allowing weights to be assigned to each of the three metrics using command line flags. Exposing these design dimensions to the autotuner also significantly reduces the complexity of the compiler implementation.

The second step of mapping consists of assigning variables to registers or shared memory. A command line flag specifies the target number of CTAs per SM which places an upper bound on the amount of shared memory and number of registers available to any one CTA. Shared memory is considered first as it is usually the more constrained resource. There are three ways shared memory can be used by a warp-specializing compiler, each of which is illustrated by a Singe kernel:

- *Store* - there are few enough values shared between operations that they can all be stored in shared memory (viscosity).
- *Buffer* - due to extremely large working sets, values are kept in registers and shared memory is only used as a buffer for communicating values between different warps (chemistry).
- *Mixed* - some values are stored in shared memory, while the remaining fraction of shared memory functions as a buffer for communication between different warps (diffusion).

The choice of where to place data and how to use shared memory may also be application or architecture dependent. Singe uses a simple working set analysis that attempts to keep values in registers unless they are accessed by many threads in which case they are assigned to shared memory. If the total number of registers needed exceeds the number budgeted by the target number of CTAs, then Singe emits a warning indicating that spilling is likely to occur. Singe also makes the choice of data placement available to the autotuner via command line flags which specify how shared memory and registers should be employed.

4.2 Named Barrier Placement and Scheduling

Traditional data-parallel GPU programming models present a very simple coarse-grain barrier for synchronization within a CTA. Named barriers provide a finer granularity of synchronization between individual warps, but can lead to poor performance and deadlock if placed incorrectly. In this section we describe an algorithm for placing named barrier synchronization calls that guarantees a deadlock-free schedule. We also present a class of transformations on the schedule that can safely be performed by the compiler during optimization passes.

In order to guarantee correct synchronization of communication between different warps we rely on the following algorithm:

1. Tag every data dependence between two operations assigned to different warps as requiring a *synchronization point*, with the producer warp needing to perform an arrival and the consumer warp needing to perform a wait.
2. Construct a partial order of the synchronization points based on their transitive data dependences.
3. Linearize the partial order of all synchronization points and then number each synchronization point, defining a total order on synchronization points.
4. For each warp, create a static schedule of all the operations in that warp that obeys operational data dependences and obeys the total ordering on synchronization points so that an operation with a lower numbered synchronization point comes before an operation with a higher numbered synchronization point.

We now prove that using this algorithm guarantees the existence of at least one schedule that is deadlock-free.

Theorem 1. A schedule for each warp that obeys the initial data dependences and a total ordering on synchronization points exists and is deadlock-free.

Proof. Initially the graph of operations is a DAG which defines a partial ordering on the operations. For every operation which requires a synchronization point we add an additional edge to every operation that requires a synchronization point with a larger number. By construction, these edges also obey the partial order of the original DAG since the total order on synchronization points was constrained by the original data dependences. Therefore a partial order still exists on the graph which guarantees that the graph is still a DAG. From any partial order, there exists at least one total ordering on all the operations. An initial schedule for a warp can be then be found by removing the operations assigned to each warp in order from any total ordering of all operations. The DAG nature of the resulting graph ensures the absence of any cycles on control resources which could result in deadlock and therefore the resulting schedule is deadlock-free. □

After an initial deadlock-free schedule has been generated for each warp, there are several re-ordering transformations permitted that do not invalidate Theorem 1.

- Operations arriving at a synchronization point can be re-ordered up the schedule arbitrarily far, or can be moved down the schedule if they are not re-ordered with respect to an operation that waits on a higher-numbered synchronization point.
- Operations waiting at a synchronization point can be re-ordered up the schedule if they are not re-ordered with respect to a lower-numbered synchronization point, or can be re-ordered down the schedule if they are not re-ordered with respect to operations on a higher-numbered synchronization point.

These two transformations on the schedule are useful for several purposes. First, they can be used to hoist operations which lie on the kernel's critical dataflow path so they are performed before less important operations. Second, using these transformation, multiple synchronization points between common sets of warps can be grouped together. This allows for bulk communication through shared memory between warps and reduces the total number of named barrier synchronizations. Singe uses scheduling heuristics to achieve both of these goals.

```
const int warp_id = (threadIdx.x >> 5);
const int lane_id = (threadIdx.x & 0x1f);
...
if ((1 << warp_id) & 0x000005D7) {
  int troe_idx = troe_index_0[0+step*step_stride][warp_id];
  double fcent = scratch[TROE_OFFSET+troe_idx][lane_id];
  double flogpr = log10(pr) - 0.4 - 0.67 * fcent;
  double fdenom = 0.75 - 1.27 * fcent - 0.14 * flogpr;
  double fquan = flogpr / fdenom;
  fquan = fcent / (1.0 + fquan * fquan);
  rr_f_0 = rr_kinf * pr/(1.0 + pr) * exp(fquan * DLn10);
} else {
  rr_f_0 = rr_kinf * pr/(1.0 + pr);
}
```

Listing 1. Overlaid bit-mask code emitted by Singe.

After the schedule for each warp has been fixed, a warp-specializing compiler must map the set of synchronization points onto the set of 16 named barriers per SM available on both Fermi and Kepler architectures[1]. The problem of mapping synchronization points onto named barriers is isomorphic to the problem of register allocation for static single-assignment (SSA) code and can be performed in polynomial time[7]. In practice, only the heptane chemistry kernel has required the use of all 16 named barriers. The final output of this stage is a forest of ASTs (one AST for each warp) which codify the chosen schedule of operations and include code to perform the necessary synchronizations.

5. Warp-Specialized Code Generation

In this section we describe techniques for emitting high-performance warp-specialized code. The primary obstacle to overcome when generating warp-specialized code is the performance of the GPU's instruction cache. GPUs are built assuming all threads run the same code with minimal stretches of control divergence. The naïve code generation strategy of using a top-level switch statement on the warp ID to send each warp to a different block of code violates this assumption and results in severe performance degradation. Figure 9 (on previous page) shows a comparison of naïve warp-specialized code with code emitted by Singe for a DME viscosity kernel over a range of warps per CTA. The naïve approach begins thrashing the instruction cache at six different warp code paths, while the code emitted by Singe continues to improve with peaks for warp counts that evenly divide the number of species in the DME mechanism. It is therefore imperative that common code paths be maintained across warps and that branching on warp IDs be done at a fine enough granularity to avoid thrashing the instruction cache.

5.1 Overlaying Computation

In order to minimize instruction cache thrashing, a warp-specializing compiler must *overlay* code from different warps whenever warps are performing similar computations. To achieve this goal we modify the standard approach to generating code from an AST. Code generation from an AST traditionally involves traversing the AST from top to bottom, emitting code at a node and then emitting code for each of a node's children in program order. In a warp-specializing compiler, code must be generated from a forest of ASTs with each AST describing the code to be executed for a different warp. To generate code from this forest of ASTs, a warp-specializing compiler traverses all the ASTs simultaneously. As long as the AST nodes across all warps are identical (with the exception of different constant values and indexing offsets, discussed in Sections 5.2 and 5.3), the compiler emits a single instance of code for all the warps in the kernel to execute[2]. When the structure of the ASTs differs, the compiler uses branches dependent on warp ID to differentiate code blocks for warps.

[1] If the desired occupancy is more than one CTA per SM, then the maximum number of named barriers per CTA is 16 divided by the desired number of CTAs per SM since named barriers are a conserved resource and, similar to shared memory and registers, can restrict occupancy.

```
__shared__ volatile double real_mirror[NUM_WARPS];
const int warp_id = (threadIdx.x >> 5);
const int lane_id = (threadIdx.x & 0x1f);
...
if (lane_id == 3)
  real_mirror[warp_id] = real_constants[0];
double arrhenius = real_mirror[warp_id] * vlntemp;
if (lane_id == 4)
  real_mirror[warp_id] = real_constants[0];
arrhenius = __fma_rn(real_mirror[warp_id], ortc, arrhenius);
```

Listing 2. Example Fermi constant broadcasts.

```
int hi_part, lo_part;
hi_part = __shfl(__double2hiint(real_constants[0]),3,32);
lo_part = __shfl(__double2loint(real_constants[0]),3,32);
double arrhenius = __hiloint2double(hi_part,lo_part) * vlntemp;
hi_part = __shfl(__double2hiint(real_constants[0]),4,32);
lo_part = __shfl(__double2loint(real_constants[0]),4,32);
arrhenius = __fma_rn(__hiloint2double(hi_part,lo_part), ortc, arrhenius);
```

Listing 3. Example Kepler constant broadcasts.

Singe uses two different approaches to branching on warp ID. In the first case, if there are no similarities between the code required for each warp, Singe emits an indirect branch dependent on warp ID to jump to the correct block of code for a given warp. For cases with longer sequences of instructions, the single indirect branch statement is fissioned into multiple indirect branch statements. In our experience this approach does not degrade performance provided the regions of code along each path of an indirect branch are less than a few hundred instructions long. Under these circumstance the instruction cache prefetching mechanism is capable of handling the inter-warp divergence.

In cases where there is still some structure shared among a subset of the warps, Singe uses bit-masks to indicate which warps should enter a block of code. Bit-masks are constructed using one-hot encoding with each bit in the mask indicating whether a warp should take the branch or not. Listing 1 shows an example of overlaid code generated by Singe for computing Laundau-Teller and Lindermann reaction rates simultaneously in the chemistry kernel. By overlaying code with bit-mask warp filters, Singe minimizes the number of different paths warps can take during execution, thereby improving performance of the GPU instruction cache.

In general we have found that branching many ways for a few hundred instructions or less, or only branching a few ways for longer stretches of code, is necessary to avoid thrashing the GPU's instruction cache. This is consistent with earlier results on using warp specialization that only contained two or three different warp-specialized code paths[3]. The penalty for thrashing the instruction cache is routinely performance degradation of an order of magnitude or worse. Therefore it is crucial that any warp-specializing compiler overlay warp-specialized code on current GPUs.

5.2 Constant Arrays and Constant Deduplication

One of the challenges in generating overlaid warp-specialized code is that often warps require different constant values. We describe a technique that avoids branching on different constant values.

After a computation has been partitioned for warp specialization, each warp requires only a subset of the total constants needed for the full computation. The DSL compiler can then allocate an array for storing the constants needed by each warp that is as large as the largest number of constants needed by any warp. The compiler emits code so that all warps access the same locations in the

[2] Care must also be taken to standardize variable names wherever possible between different warps to avoid creating false AST differences.

Mechanism	Viscosity	Diffusion	Chemistry
DME	8	18	6
Heptane	28	28	8

Figure 10. Constant registers per thread on Kepler.

constant array at all times. After code generation is complete, the compiler lays out constants in memory to ensure the right values are in the correct locations of the array for each warp. In some cases for divergent code, this may involve emitting padding values into the array for some warps. However, the cost of loading a few padded values is small compared to the cost of dynamic branching.

In practice, we have discovered that these constant arrays are often quite large for a single warp and can consume more than an entire thread's worth of registers. We therefore have developed another optimization for deduplicating constants between the threads within a warp. All the threads within a warp require the same set of constants and a single thread can broadcast a constant value to the other threads within a warp with no synchronization. Instead of each thread holding all the constants required for computation, the constants for a warp are statically striped across the threads within a warp, requiring each thread hold only $\frac{1}{32}$ of all the constants. Whenever a constant is needed, it is broadcast from the owning thread to the other threads within the warp.

This broadcast takes different forms depending on the architecture. For Fermi GPUs, the broadcast is performed using an allocation of shared memory with one location per warp. One thread writes data into shared memory and then the other threads in the warp read the value; no explicit synchronization is required because all the threads within a warp execute in lock step. Listing 2 shows example code emitted by Singe employing this technique.

On Kepler GPUs the broadcast is performed more efficiently using *shuffle instructions*. Shuffle instructions allow an exchange of 32 bits between all threads in a warp. By breaking double precision constants into their upper and lower halves they can be broadcast from the owning thread and then re-assembled. The two shuffle instructions are faster on Kepler because they do not stall pipelined loads in the shared memory pipeline, which happens during the shared memory write for the Fermi broadcast. Listing 3 shows an example of exchanging constants using shuffle instructions.

Figure 10 shows the number of registers required per thread for storing constants values across both mechanisms on the Kepler architecture. Note that they are small enough to allow the majority of registers to remain free for general purpose computation while still holding more constants on chip than can even be addressed by the constant cache, let alone fit in its working set[1].

This approach to storing constant values in registers can yield further performance gains when coupled with a streaming execution model. When multiple sets of points are mapped onto a single CTA, the CTA performs a loop to handle all of its points. By hoisting the loads for the constants outside of this loop, a DSL compiler can amortize the cost of loading constants into registers, resulting in very low overhead constant access.

5.3 Warp Indexing

For applications with irregular memory access patterns, warp specialization can result in warps needing to access different locations in memory. One of the many examples of this occurring in Singe is in the stiffness computations for the chemistry kernel described in Section 3.4. Each of the different warps needs to load different diffusion rates from global memory and different molar fractions from shared memory. The need to access different memory locations by different warps runs counter to the DSL compiler's goal of overlaying as much code as possible. To support code overlay without dynamic branching, a compiler can use *warp indexing* constants when doing memory accesses.

```
1 __shared__ volatile double scratch[192][32];
2 const int lane_id = threadIdx.x & 0x1f;
3 ...
4 int index = __shfl(index_constants[0],1,32);
5 asm volatile("ld.global.nc.cg.f64 %0,_[%1];" : "=d"(stif_diffusion_0) :
6   "l"(diffusion_array+index*spec_stride) : "memory");
7 stif_mole_frac_0 = scratch[index][lane_id];
```

Listing 4. Stiffness warp indexing on Kepler.

Warp indexing is a technique where each warp stores integer offset values for indexing into memory that are specific to that warp. All warps can then perform their address calculations using the same index variable even though the variable stores different values for different warps. As with the constant arrays, this extra level of indirection enables the compiler to overlay warp-specialized code without requiring dynamic branching for irregular memory access patterns.

Similar to constant deduplication, if the number of warp indexing constants is large, they can be deduplicated by striping them across the threads within a warp using the same approach described in Section 5.2. The only difference is that the index constants are not directly involved in computation, but are instead only used for address indexing. Listing 4 shows an example of warp indexing on Kepler from the stiffness computation where the same index is used to load a diffusion value from global memory and a molar fraction from shared memory.

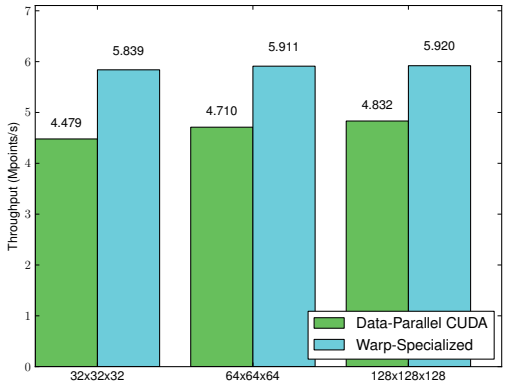
One important caveat to warp indexing is that it only applies to data that is stored in dynamically indexable GPU memories, specifically global, shared, and constant memories. Arrays allocated in registers are not dynamically indexable and any attempt to index them with a dynamic variable causes the CUDA compiler to spill the array to much slower local memory. Ideally future GPUs will allow dynamic indexing of the register file as well.

6. Experimental Results

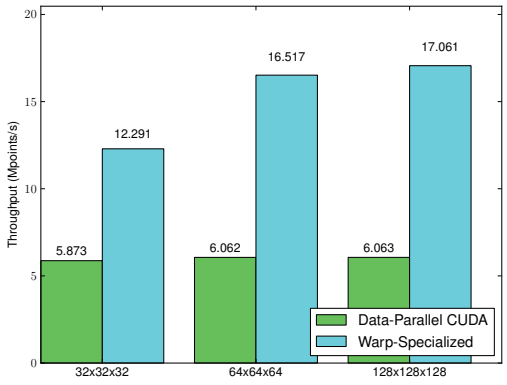
In this section we quantify the performance gains that result from applying warp specialization in Singe. We compare against baseline versions of the kernels emitted by an earlier version of Singe in the traditional CUDA data-parallel programming model. The baseline versions of the kernels were fully optimized using a combination of domain- and architecture-specific optimizations, including the use of logarithmic-space computations, exposure of additional instruction level parallelism, and the use of LDG texture loads using inline PTX for higher memory bandwidth on the Kepler architecture. In addition, a brute-force autotuner exhaustively explored the space of occupancy-register tradeoffs on all architectures. Consequently, the baseline CUDA kernels were already up to 2X faster than the OpenACC versions currently used by the production version of S3D[11]. The same set of optimizations were also applied to the warp-specialized versions of the kernels to ensure that all performance gains are directly attributable to warp specialization.

All experiments were run on two different architectures. The first was an NVIDIA Tesla C2070 Fermi GPU with 14 SMs, a 1147 MHz SM clock frequency, and a 1494 MHz DRAM clock frequency. The second architecture was a Tesla K20c Kepler GPU with 13 SMs, a 705 MHz SM clock frequency, and a 2600 MHz DRAM clock frequency. ECC was disabled on both architectures to make the memory-bound baseline kernels perform as well as possible against the kernels emitted by Singe. Singe kernels are primarily limited by on-chip resources and therefore achieve even larger speedups relative to the baseline kernels with ECC enabled. All kernels were compiled with the default version of `nvcc` from CUDA 5.0 and were run with version 304.54 of the CUDA driver.

For all experiments, we report throughput for three different grid problem sizes 32^3, 64^3, and 128^3 to illustrate any scaling effects. Absolute times for each experiment can be computed by di-

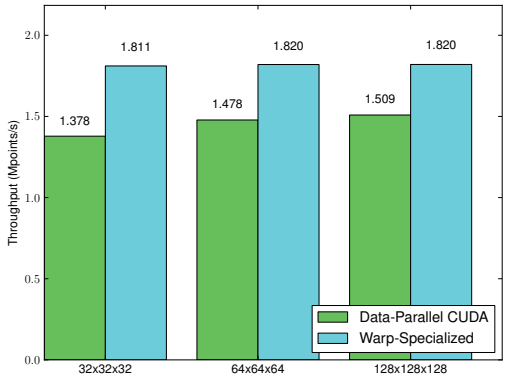


(a) Fermi

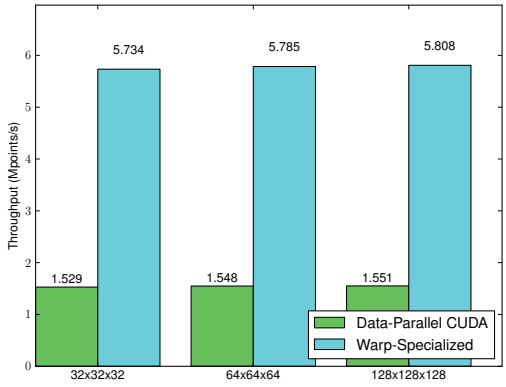


(b) Kepler

Figure 11. Viscosity performance results for DME.



(a) Fermi



(b) Kepler

Figure 12. Viscosity performance results for heptane.

viding the number of points in each grid by the reported throughput. In all cases the reported throughput is computed using the harmonic mean of the throughputs for twenty iterations of each experiment. In addition to reporting performance results for each kernel, we also analyze the underlying SASS machine code for each kernel to determine the limiting factor for each of the different kernels.

6.1 Viscosity

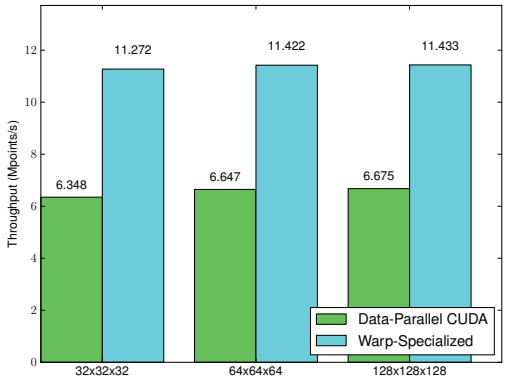
Figures 11 and 12 show performance results comparing warp-specialized versions of the viscosity kernel to the baseline versions for the DME and heptane mechanisms respectively. Speedups for the warp specialization kernels ranged from 1.2X to 3.75X over the baseline kernels. Significantly larger speedups over the baseline versions were achieved on the Kepler architecture.

To understand the underlying reasons for these results we investigated the underlying SASS machine code generated for both the baseline and warp-specialized versions of the code. From the SASS we were able to determine the total number of double precision floating point operations required per warp for both the baseline and warp-specialized versions on Fermi and Kepler. For the DME mechanism on Fermi the baseline and warp-specialized kernels achieved 197.9 and 257.3 billion double precision floating point operations per second (GFLOPS) respectively; on Kepler they achieved 220.6 and 617.7 GFLOPS respectively.

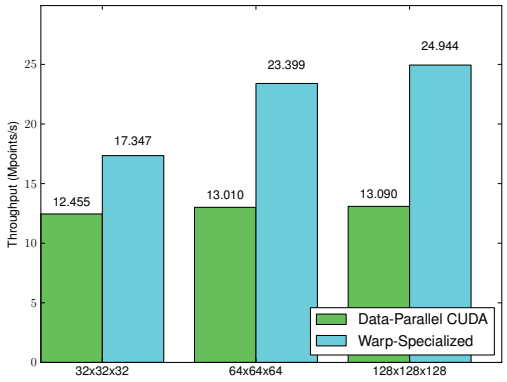
Using these numbers along with an understanding of the Fermi and Kepler architectures we were able to discern the limiting factors for each kernel. In theory, a Fermi GPU can issue 1 DFMA instruction per SM every other clock cycle. For the C2070 GPU in these experiments this yields a theoretical math throughput of 513 GFLOPS. In practice, optimized Fermi kernels such as DGEMM can reach around 300 GFLOPS[1]. Achieving 257.3 GFLOPS on the warp-specialized viscosity kernel is near optimal math throughput on Fermi when the overhead of dynamic branching and shared memory accesses are included. The baseline CUDA version however does not come as close to the practical peak math throughput for two reasons. First, the working set of molar fractions and per-species viscosities does not fit in registers and is spilled to local memory. Second, the large number of constants do not fit in the constant cache. Both these issues add latency and account for the slowdown relative to the warp-specialized kernels.

For the Kepler architecture, the theoretical math throughput is significantly higher. On Kepler each *quad* of an SM can issue one DFMA every other cycle, yielding a theoretical throughput of 1173 GFLOPS on a K20c with 13 SMs. The 617.7 GFLOPS achieved by the warp-specialized viscosity kernel is more than half of theoretical peak. We hypothesized that this kernel is actually limited by the throughput of DFMA operations whose third operand is loaded from the constant cache, which is the case for the 12 DFMA operations in the Taylor series expansion of the exponential function which dominates viscosity performance. To verify this hypothesis we modified Singe to emit an incorrect exponential function that instead relied on constants stored in registers. Experiments showed these kernels were capable of performance near 750 GFLOPS, indicating that our warp-specialized viscosity kernels were also compute-bound on Kepler. The baseline CUDA version improved marginally from Fermi, but still suffered from severe register spilling and constant cache misses. The highest performing baseline CUDA version used the larger on-chip register file on Kepler to avoid spilling and not to increase occupancy which meant that the latency of loading constants was still exposed.

These results demonstrate that warp specialization enables Singe to partition the viscosity kernel in a way that better maps

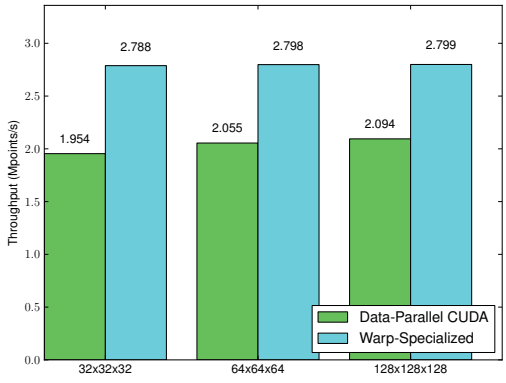


(a) Fermi

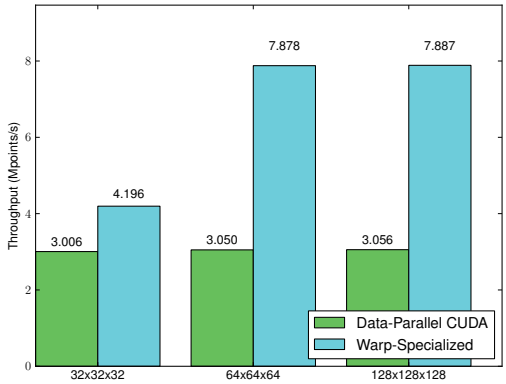


(b) Kepler

Figure 13. Diffusion performance results for DME.



(a) Fermi



(b) Kepler

Figure 14. Diffusion performance results for heptane.

onto both Fermi and Kepler GPUs. Both warp-specialized kernels approach the limits of math throughput on their architectures. The higher potential math throughput on Kepler explains why the Kepler warp-specialized kernel performs significantly better than the Fermi warp-specialized kernel relative to the baseline.

6.2 Diffusion

Figures 13 and 14 show performance results for the diffusion kernels for the DME and heptane mechanisms respectively. Speedups for the warp-specialized kernels ranged from 1.33X to 2.58X over the baseline CUDA kernels. Again larger speedups were achieved on the Kepler architecture than the Fermi architecture due to the higher ceiling of math instruction throughput.

To confirm that the performance of the warp-specialized kernels was again limited by math throughput, we again analyzed the SASS machine code. We were surprised to discover that for the DME mechanism the warp-specialized version of the kernel was only achieving 212.8 GFLOPS on Fermi and 526.6 GFLOPS on Kepler. After examining the SASS, we determined that the degradation in performance was caused by the additional named barrier calls that were needed to synchronize access to the partial sums stored in shared memory. A larger number of barriers resulted in excessive cycles waiting for straggler warps to arrive at the barrier which consequently reduced math throughput. Unsafely removing the barriers resulted in performance around 250 GFLOPS on Fermi and 625 GFLOPS on Kepler, which is consistent with the actual peak math throughput observed in Section 6.1.

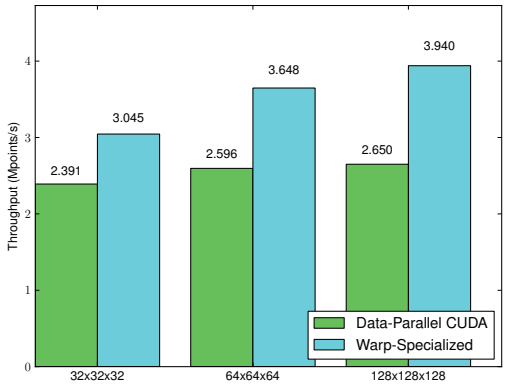
Another interesting effect can be observed in Figures 13(b) and 14(b). For smaller problem sizes the speedup over the baseline kernels is smaller. Due to the large number of constants required for the diffusion kernel, the cost of loading these constants into registers is significant. However, for larger problem sizes it is more easily amortized. This effect is only noticeable on Kepler where the math performance is much higher and exposes the constant loading phase. A problem size of at least 64^3 is therefore required to fully amortize the cost of loading the constants into registers.

6.3 Chemistry

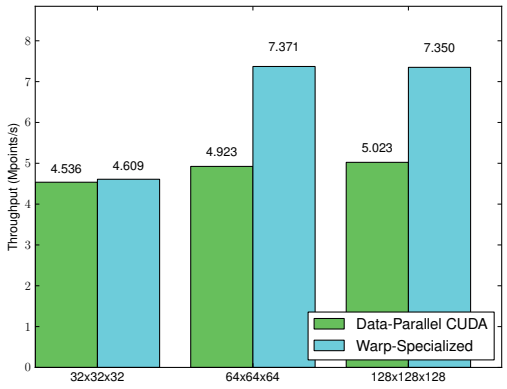
Figures 15 and 16 show performance results for the chemistry kernels for both the DME and heptane mechanisms respectively. Speedups for the warp-specialized kernels ranged from 1.01X to 1.50X over the baseline CUDA kernels. Unlike the viscosity and diffusion kernels, the performance characteristics of the chemistry kernels are very different. The very large working set places extreme pressure on the architectural resources of any GPU. Using warp specialization Singe was able to emit code using an alternative allocation of resources which yielded performance gains.

The baseline CUDA chemistry kernels spill significant amounts of memory due to the large working sets required. For the heptane mechanism 8736 bytes are spilled per thread on Fermi and 8500 bytes per thread are spilled on Kepler. With enough occupancy all this latency can be hidden, but results in a memory bandwidth limited kernel. After analyzing the SASS for the baseline CUDA kernels, we measured memory bandwidths of 85 GB/s on Fermi and 100 GB/s on Kepler, which are consistent with actual peak memory bandwidths for these architectures[3].

[3] We have measured read bandwidths of 165 GB/s on a Kepler K20c with five GDDR5 memory partitions when using LDG texture loads, but this path is not available to local memory since local memory is not constant. 100 GB/s load bandwidth is consistent with measured load bandwidths through the much smaller pipe in the L1 cache with only 13 SMs.

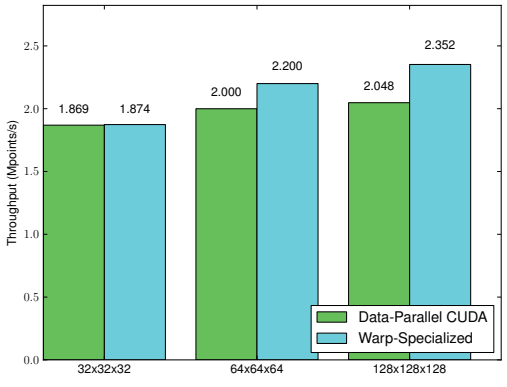


(a) Fermi

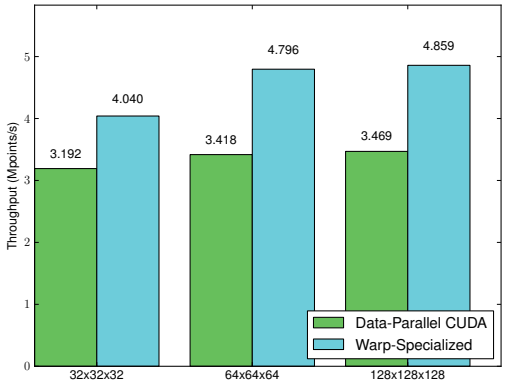


(b) Kepler

Figure 15. Chemistry performance results for DME.



(a) Fermi



(b) Kepler

Figure 16. Chemistry performance results for heptane.

Alternatively, the warp-specialized kernels emitted by Singe for heptane spill only 276 bytes on Fermi and 44 bytes on Kepler (mostly infrequently accessed pointer data). Almost the entire working set of the computation including all forward and reverse reaction rates remain in registers at the cost that they must be repeatedly exchanged through shared memory. Neither the Fermi nor the Kepler warp-specialized kernels have enough warps to hide the 30 cycles of shared memory access latency with only 20 and 16 warps per SM respectively. As a result, the warp-specialized versions of the kernel are shared memory latency limited. However, the performance results demonstrate that this is significantly better than being global memory bandwidth limited.

7. Discussion

While we have shown that warp specialization is effective on NVIDIA GPUs, we believe that similar benefits could be realized on other wide-vector SIMD architectures, such as Intel's Xeon Phi. Conceptually, a warp is a single instruction stream issuing 32-wide vector instructions, which is not far removed from a thread on a Xeon Phi issuing 8-wide double-precision vector instructions. Instead of partitioning and mapping computations onto warps, computations could be mapped onto vectorized threads. Identical to warp specialization, the techniques presented in this paper could be used to fit large working sets into on-chip memories and handle irregularity. There are currently two absent hardware features on the Xeon Phi that inhibit this approach. First, there are no fast hardware synchronization primitives equivalent to named barriers present (heavy-weight software mutexes are currently necessary). Second, there are no on-chip software-managed memories to be used for fast data exchange buffers. Instead the hardware-managed L1 cache must be used which is subject to interference and eviction by other data movement operations.

The presence of hardware-managed caches on GPUs also hinders the performance of warp-specializing compilers like Singe. Hardware-managed caches consume transistors which could have been used to fit larger working sets in on-chip software-managed memories. Hardware-managed caches are also difficult resources for compilers to reason about. An earlier version of Singe attempted to store the large number of constant values in the GPUs hardware-managed L2 cache. However, writes from other threads kept evicting constants off-chip despite Singe annotating all global writes with the PTX cache-streaming qualifier `cs`. The current Singe compiler completely ignores the L2 cache which wastes significant on-chip memory (768 KB on Fermi and 1536 KB on Kepler). Finally, hardware-managed caches add overhead to the kernels emitted by Singe both in terms of performance (tag look-ups on all memory accesses) and power (unnecessary data movement between cache levels). Ideally, future GPUs and other wide-vector architectures will consist primarily of software-managed caches which will enable compilers like Singe to directly orchestrate data movement leading to both higher performance and power efficiency.

The poor performance of current GPU instruction caches makes warp specialization infeasible without overlaying code paths. Instruction caches designed for handling many divergent warps in future GPUs would remove the need for overlaying code using the techniques described in Section 5. This would improve the performance of Singe kernels by significantly reducing the number of necessary warp-specific branch instructions. Additionally, it would greatly reduce the complexity of the code generation stage for warp-specializing compilers. It would also make writing warp-specialized code practical for human programmers.

Finally, a common misconception regarding warp specialization is that it will be made obsolete by Moore's law because more transistors in future chips will allow current large working sets to fit on-chip. Ideally, more on-chip storage could lead to higher occupancy and would only require a simple data-parallel programming model to achieve high performance. Unfortunately, this view fails to consider that many computational science domains, including combustion, cosmology, and molecular dynamics, all limit the scope of their simulations in order to execute efficiently on current hardware. For example, the combustion mechanisms described in this paper are considered *reduced* mechanisms because they only simulate tens of chemical species instead of the hundreds and thousands of species in real mechanisms[12]. In our experience, computational scientists will always use additional hardware to perform higher fidelity but more computationally expensive simulations instead of making current simulations perform better. As machines become more powerful, scientists will devise more complex simulations that place intense pressure on both programming systems and hardware. Under such conditions, data-parallel programming models such as CUDA, OpenCL, and especially OpenACC will struggle to take full advantage of future hardware. Ultimately, programming models such as warp specialization will become essential for managing large working sets, handling irregularity, and exploiting task-level parallelism to fully leverage forthcoming architectures for general purpose computing.

8. Related Work

The most closely related work to Singe is CudaDMA[3], a library that uses warp specialization to optimize data movement between on-chip and off-chip memories. CudaDMA kernels specialize warps into *compute warps* for application code, and *DMA warps* for data movement. Synchronization is performed via a simple interface using named barriers. There are many differences between CudaDMA and Singe. For example, with only two warp code paths, CudaDMA programs do not require Singe's optimizations to avoid instruction cache thrashing. As another example, all CudaDMA kernels use the same partitioning scheme: application code runs on compute warps and CudaDMA code executes on DMA warps. Avoiding the partitioning problem faced by Singe reduces the implementation complexity of CudaDMA. The focus is also different: CudaDMA is a general purpose library for CUDA users, while the techniques presented here are for warp-specializing compilers.

Green-Marl is a DSL and compiler for implementing graph analyses[8] on GPUs. The Green-Marl compiler makes use of the concept of virtualized warps to handle the irregularity inherent in graph algorithms. While the creation of virtualized warps is similar to the partitioning operation required for warp specialization, the Green-Marl compiler maps virtualized warps onto physical warps to conform to the data-parallel GPU programming model.

G-Streamline is a framework that dynamically detects computational and memory access irregularities in GPU kernels[14]. Using this information G-Streamline rewrites kernels on the fly to reduce the overheads associated with irregularity. In this work we've shown how warp-specializing DSL compilers like Singe can handle irregularity without any dynamic overheads.

Halide is a high-level DSL and autotuning framework for two dimensional image processing pipelines[13]. Halide performs many interesting optimization techniques for taking advantage of various GPU resources similar to how a DSL compiler might leverage warp specialization. However, Halide does not consider warp specialization as an approach for optimizing GPU programs.

There have been many DSL languages that target GPUs for high performance[4, 6, 8]. To the best of our knowledge, we are the only ones who demonstrate the necessary techniques for constructing a warp-specializing DSL compiler.

9. Conclusion

We have introduced warp specialization as an effective compiler technique for generating GPU code for applications with both irregular computation and memory accesses as well as large working sets. We have also presented Singe, a DSL compiler that leverages warp specialization in conjunction with domain specific knowledge to produce GPU combustion kernels that perform better than is possible in standard data-parallel GPU programming models. We described the general architecture and compilation techniques necessary for constructing a warp-specializing DSL compiler. Using warp specialization, Singe emits kernels that are up to 3.75X faster than previously optimized but purely data-parallel GPU code.

Acknowledgments

This work was supported by the ExaCT Combustion Co-Design Center via Los Alamos National Laboratory Subcontract No. 173315-1 through the U.S. Department of Energy Office of Advanced Scientific Computing Research under Contract No. DE-AC52-06NA25396. The authors thank the anonymous reviewers, our shepherd John Reppy, Elliott Slaughter, Zach DeVito, Michael Garland, Bryan Catanzaro, and Pat McCormick for their useful comments and suggestions. Special thanks go to Jackie Chen, Hemanth Kolla, and Tianfeng Lu for their detailed explanations and patience in answering questions regarding combustion science.

References

[1] CUDA programming guide version 5.5. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2013.

[2] Parallel thread execution ISA version 3.2. `http://docs.nvidia.com/cuda/parallel-thread-execution/index.html`, 2013.

[3] M. Bauer, H. Cook, and B. Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. SC '11, 2011.

[4] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. PPoPP, pages 35–46, 2011.

[5] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, 2009.

[6] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. SC, pages 9:1–9:12, 2011.

[7] S. Hack and G. Goos. Optimal register allocation for SSA-form programs in polynomial time. *Inf. Process. Lett.*, 2006.

[8] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. PPoPP, 2011.

[9] R. Kee, F. Rupley, and E. Meeks. CHEMKIN-III: A fortran chemical kinetics package for the analysis of gas-phase chemical and plasma kinetics. 1996.

[10] Khronos. The OpenCL Specification, Version 2.0. `http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`, 2013.

[11] J. M. Levesque, R. Sankaran, and R. Grout. Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond. SC '12, pages 15:1–15:11, 2012.

[12] T. Lu and C. K. Law. Toward accommodating realistic fuel chemistry in large-scale computations. *Progress in Energy and Combustion Science*, pages 192 – 215, 2009.

[13] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 2012.

[14] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dyn. irregularities for GPU computing. ASPLOS, 2011.