Efficient Compilation of Fine-Grained SPMD-threaded Programs for Multicore CPUs

John A. Stratton[†*] Vinod Grover[†]
Jaydeep Marathe[†] Bastiaan Aarts[†]
Mike Murphy[†] Ziang Hu[†]
[†]NVIDIA Corporation
{vgrover, jmarathe, baarts, mmurphy, zhu}
@nvidia.com

Wen-mei W. Hwu[*]
[*] University of Illinois at Urbana-Champaign,
Center for Reliable and High-Performance
Computing
{stratton, hwu}@crhc.illinois.edu

Abstract

In this paper we describe techniques for compiling fine-grained SPMD-threaded programs, expressed in programming models such as OpenCL or CUDA, to multicore execution platforms. Programs developed for manycore processors typically express finer thread-level parallelism than is appropriate for multicore platforms. We describe options for implementing fine-grained threading in software, and find that reasonable restrictions on the synchronization model enable significant optimizations and performance improvements over a baseline approach. We evaluate these techniques in a production-level compiler and runtime for the CUDA programming model targeting modern CPUs. Applications tested with our tool often showed performance parity with the compiled C version of the application for single-thread performance. With modest coarse-grained multithreading typical of today's CPU architectures, an average of 3.4× speedup on 4 processors was observed across the test applications.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Algorithms, Performance

Keywords CUDA, Multicore, CPU, SPMD

1. Introduction

In the coming years, commercial application developers will have a strong incentive to develop highly parallel software to take advantage of widespread parallel processors in the consumer market. However, it is unclear whether each potential user of an application will have a computing subtrate with a similar degree, granularity and style of parallelism. Even if an application is amenable to targeting a wide variety of parallel computational platforms, it is unclear whether a single expression of the application in any one programming model will be sufficient. The model must be powerful enough to effectively capture many applications, yet have enough constraints to enable a wide range of architectures to be effectively supported.

We present some initial findings of a case study testing one parallel programming model that industry is hoping will be such a portable model: fine-grained Single Program Multiple Data (SPMD) kernels, with limited thread cooperation, controlled by a centralized process. CUDA [16] and OpenCL [12], for example, are both built on an underlying programming model of fine-grained SPMD threads. For the experiments presented here, we will be working with the CUDA programming model, noting in advance that the same techniques would be applicable to OpenCL and other SPMD programming models as well.

The CUDA programming model is a hybrid of two parallel programming models initially tailored to GPU architectures. It supports bulk synchronous task parallelism [24], where each task is composed of fine-grained SPMD threads. Programmers have been using CUDA with significant success in many application fields, such as bioinformatics [19], molecular dynamics [21], machine learning [4], and medical imaging [22]. We view these successes as sufficient evidence that the fine-grained SPMD model is effective for programming a manycore architecture with explicit support for fine-grained threads. However, previously there has not been investigation of how such a model could effectively map to a more coarsely threaded architectures such as the current commodity multicore processors.

The contributions of this paper are:

- An implementation and comparison of two approaches to implementing a fine-grained SPMD programming model on a processor with coarse-grained thread-level parallelism.
- A description of programming model restrictions necessary to implement the intuitively more effective approach.
- Optimizations enabled by the serialization of a parallel model, primarily redundancy removal in both computation and data storage.
- Experimental evidence confirming the intuition, and comparing it with standard compiled C on current multicore CPUs.

The primary enabling factors for generating efficient C code from a fine-grained threading model are the restrictions on synchronization usage. These restrictions allow stronger reasoning in the compiler about execution semantics in the static code. The baseline microthreading approach to serializing an SPMD programming model is described in Section 4. The baseline approach represents what we believe to be the state of the art in implementing general finely-threaded programs on a system with significantly less thread-level parallelism. The second approach is summarized in our own previously published work [23] and that of Shirako et al. [20], describing a basic approach for generating structured code serializing fine-grained SPMD code. We have reimplemented and extended the functionality of these algorithms within a production-level compiler, and compile the full CUDA language without the limitations of the previous work. We show with experimental results that the structured approach enabled by restrictions on synchronization usage does indeed provide significant performance benefits over the more general baseline.

In the context of a serialized parallel model, several optimizations not available to the parallel form of the code are enabled. The optimizations detailed in Section 6 are notably analogous to existing redundancy removal optimizations in sequential programming models. However, we can leverage knowledge of explicit parallelism to reduce the burden of analysis or surpass the typical capabilities of commercial implementations.

We highlight some of the related work in cross-architecture parallel programming models in Section 2. A concise description of CUDA's execution and memory models relevant to this work is presented in Section 3. The general microthreading and structured microthreading techniques are discussed in Sections 4 and 5 respectively, followed by a description of enabled optimizations in Section 6. We describe the practical details of our compiler and runtime environment in Section 7 to provide a full context for our performance results presented in Section 8. We summarize the experiments and lessons learned in the concluding remarks of Section 9.

2. Related Work

The issue of mapping small-granularity parallel work units to CPU cores has been addressed in other programming models, such as parallel simulation frameworks [7] and dataflow or message-driven programming models [2, 3]. Such models typically implement a user-level microthreading technique similar to our baseline approach. Microthreading implementation is simplified when implemented within a single code object, as an SPMD programming model provides. OpenCL [12] is a programming model closely related to CUDA that claims such platform portability as we would like to explore. However, it has not matured to demonstrate such portability at this time. The methods and results presented here would be directly applicable to all finely-threaded SPMD programming models, including OpenCL.

Shirako et al. [20] applied many of the same transformation methodologies to serialize data-parallel loops containing barriers. We demonstrate how similar techniques can be utilized in an SPMD programming model, and demonstrate the further optimizations enabled by the application of these techniques.

Numerous other frameworks and programming models have been proposed for data-parallel applications for multiprocessor architectures. Some examples include OpenMP [17] and HPF [11]. Although widely used in a CPU symmetric multiprocessor environment, these models are yet to be proven for manycore chips. Lee et al. have described a system for compiling OpenMP programs to CUDA [13] which, if successful, could provide similar experimental benefit as extending CUDA to CPUs.

```
__global__ small_mm_list(float* A_list, float* B_list,
                                 , const int size)
{
  float sum;
  int matrix_start, col, row, out_index, i;
  matrix_start = blockIdx.x * size * size;
  col = matrix_start + threadIdx.x;
  row = matrix_start + (threadIdx.y * size);

  sum = 0.0;

  for(i = 0; i < size; i++)
    sum += A_list[row + i] * B_list[col + (i*size)];

  //  Barrier before overwriting input data
  __syncthreads();

  out_index = matrix_start +
                 (threadIdx.y * size) + threadIdx.x;
  A_list[out_index] = sum;
```

Figure 1: Multiplying many small matrices in CUDA.

Diamos has implemented a binary translation framework from GPU binaries to x86 [10]. While binary translators have advantages in knowing statically unavailable runtime parameters, compilers have more high-level program information available to them in the structured and symbolic source code. It is unclear which of the high-level transformations we propose would be possible without high-level compiler information available, if any.

Liao et al. designed a compiler for efficiently mapping the stream programming model to a multicore CPU architecture [14]. Their implementation attempted to build into the compiler capability for removing many of the restrictions of the stream programming model. In many ways, fine-grained SPMD-threaded models remove from the stream programming model those same limitations addressed by Liao et al.'s compiler. The programmer has control over tiling and kernel merging optimizations, the range of which is potentially broader than can be discovered and applied in an automated framework.

NVIDIA has released a toolset for CUDA program emulation on a CPU, designed for debugging. In the emulation framework, each fine-grained thread is executed by a separate runtime OS thread, incurring significant thread-scheduling overhead, and performing orders of magnitude more poorly than any of our approaches in informal experiments.

3. CUDA Programming Model

CUDA as a programming model has several interacting constructs for composing parallel programs on a shared-memory system [16]. The programming model allows sequential code in the standard C language with library APIs to control and manage *grids* of parallel execution specified by *kernel* functions. The *host* portion of the code is compiled using traditional methods and tools, while the kernel code introduces constructs for expressing SPMD parallelism. This work primarily focuses on the compilation and execution of the parallel kernel functions. We will be using the example kernel function of Figure 1 throughout this paper.

Within the SPMD kernel functions, *threads* are distinguished by an implicitly defined 3-tuple index uniquely iden-

```
tid = threadIdx.x;
while(i < end)
{
  x += input[i];
  if(i == end-1) {
    //segmented circular shift
    data[(tid + 1) % shift] = x;
    __syncthreads();
    output = data[tid];
    break;
  }
  else {
    i++;
  }
}
```

(a) Incorrect Usage

```
tid = threadIdx.x;
while(i < end)
{
  x += input[i];
  if(i == end-1) {
    break;
  }
  else {
    i++;
  }
}
//segmented circular shift
data[(tid + 1) % shift] = x;
__syncthreads();
output = data[tid];
```

(b) Correct Usage

Figure 2: Synchronization within control flow. (b) shows code semantically equivalent to that of (a), and obeys the synchronization usage constraints.

tifying threads within a thread *block*. Thread blocks themselves are distinguished by an implicitly defined 2-tuple variable. The ranges of these indexes are defined at runtime by the host code in special kernel invocation syntax. In the example of Figure 1, each thread block is computing one small matrix multiplication out of the list, while each thread is computing one element of the result matrix for its block.

CUDA guarantees that threads within a thread block will be live concurrently, and provides constructs for threads within a thread block to perform fast barrier synchronizations and local data sharing. Distinct thread blocks within a grid have no ordering imposed on their creation or execution. Atomic operations provide limited interblock communication.

CUDA uses textually-aligned static barrier semantics, such as those of the Titanium language [1]. For instance, it is illegal to invoke a barrier intrinsic in both paths of an if-else construct when CUDA threads may take different branches of the construct. Although all threads within a thread block will reach one of the intrinsics, they represent separate barriers, each requiring that either all or none of the threads reach it.

As a more general example, consider the constructed example of Figure 2. We assume that `end` is a function of the thread index, while the initial value of `i` is thread-invariant. Although each logical thread will hit the barrier exactly once, the code of Figure 2a will have unpredictable runtime behavior. Figure 2b shows how the code may be restructured to achieve the desired effect without violating this constraint.

CUDA is less restrictive than Titanium in that barriers can be dependent on statically thread-dependent expressions. It only requires that the dynamic evaluation of those expressions results in a uniform boolean value at runtime. For instance, if `end` and the initial value of `i` are functions of the thread index such that `(i - end)` is thread-invariant, the code of Figure 2a will function correctly, in constrast with the restrictions of Titanium that would prohibit this case as well.

The CUDA memory model, at the highest level, separates the host and device memory spaces, such that host code and kernel code can only access their respective memory spaces directly. The device memory spaces are the *global*, *constant*, *local*, *shared*, and *texture* memory spaces. A summary of the memory spaces is given in Table 1.

```
1   __global__ small_mm_list(float* A_list, float* B_list,
                             const int size)
    {
2     float sum[];
3     int matrix_start[], col[], row[], out_index[], i[];
      int current_restart, next_restart;
      next_restart = 0;
      // Loop over barrier synchronization intervals
      while (next_restart != -1) {
        current_restart = next_restart;
        //Loop over threads within an interval
        for(each tid) {
          switch (current_restart) {
            case 0:
              goto RESTART_POINT_0;
            case 1:
              goto RESTART_POINT_1;
          }

          // Original program beginning:
         RESTART_POINT_0:
4         matrix_start[tid] = blockIdx.x * size * size;
5         col[tid] = matrix_start[tid] + tid.x;
6         row[tid] = matrix_start[tid] + (tid.y * size);

7         sum[tid] = 0.0;

8         for(i[tid] = 0; i[tid] < size; i[tid]++)
9           sum[tid] += A_list[row[tid] + i[tid]] *
                        B_list[col[tid] + (i[tid]*size)];

          // restart point induced by syncthreads()
10        next_restart = 1;
          goto end_of_thread_loop;
         RESTART_POINT_1:
11        out_index[tid] = matrix_start[tid] +
                           (tid.y * size) + tid.x;
12        A_list[out_index[tid]] = sum[tid];
          next_restart = -1; // indicates "return"
         end_of_thread_loop:
        }
      } // while
    }
```

Figure 3: Microthreaded code for our example kernel

These memory spaces follow general microarchitecture principles. Large memory spaces are expected to have long latencies and limited random-access bandwidth, while small memory spaces can reliably satisfy low-latency accesses. Efficient CUDA programs make these cost trade-offs explicitly by using localized access patterns and limiting the active working set. However, if an application is written assuming significant hardware acceleration of texture processing operations, it could lead to design choices that perform poorly on processors implementing those features in software.

4. Baseline SPMD Microthreading

The term *microthreading* describes software techniques used in contexts where parallel work units are too small to efficiently schedule individually [2, 7]. The key concept is that software emulates the execution of multiple conceptually parallel threads or computation objects in a single, sequential program. The result of applying such a microthreading technique to the kernel of Figure 1 is shown in Figure 3. Note that the implicitly defined variable `threadIdx` has been shortened to `tid` for brevity. The compiler begins by labeling each barrier with a unique number, re-

Table 1: CUDA Device Memory Spaces in GPU Execution Context

Memory Space	Permissions	Scope of an Object	Capacity	Latency	Special Features
Global	Read/Write	All threads	DRAM capacity	High	Requires aligned, contiguous simultaneous accesses for best bandwidth.
Constant	Read-Only	All threads	64KB	Low (cached)	Single-banked cache with broadcast capability to multiple threads.
Local	Read/Write	Single thread	DRAM capacity	High	Most often promoted to private registers, which are shared between threads. Values not promoted to registers have long latency access.
Shared	Read/Write	Single thread block	16KB	Low	Scratchpad memory shared between thread blocks. More shared memory used per thread block means fewer thread blocks can be simultaneously active.
Texture	Read-Only	All threads	DRAM capacity, limits per object	High	Hardware interpolation, indexable by real-valued indexes, and other features for image processing.

serving the number zero for the implicit barrier at the beginning of the program. In our example, the single barrier gets labeled with the number 1. The original code for the program is modified, with each barrier replaced by a unique label, an assignment of the `next_restart` variable with the barrier's ID, and a jump to begin executing the next conceptual thread. All exit points from the function are replaced by statements assigning an exit flag (-1) to the `next_restart` variable. The compiler then generates the microthreading iteration structures. The master while-loop iterates over the number of times the threads will synchronize, each time updating the current restart point to the place the threads synchronized. A for-loop iterates over thread indexes, and uses a switch structure to begin each thread's execution at the current restart point. For each iteration of the conceptual thread for-loop, a single conceptual thread is advanced from its previous synchronization point to its next synchronization point. The master while-loop then iterates again to emulate all conceptual threads executing the original program from the barrier statement to the next point of syncronization, unless the original program end was reached by the conceptual threads being emulated.

In our example, the master while-loop control structure will begin executing the SPMD code of the original parallel program, marked by `RESTART_POINT_0`. The program executes the original, SPMD source code until it reaches statement 10, the original synchronization point. It then marks the synchronization point it reached, and program execution continues with the next conceptual thread at the original program beginning (statement 4). When all intances of conceptual threads have been iterated over (`each tid` is exhausted), the barrier is marked as the next restart point. This corresponds to the release of all conceptual threads from the barrier, so each microthread is executed again starting at the barrier release. Each conceptual thread then writes its output and reaches the original function's end. When all conceptual thread indexes have been processed again, the master while-loop detects that all conceptual threads have completed, and exits the function.

The memory model must also be adapted to fit a monolithic shared memory system. The globally visible memory regions already fit this model, and need not be changed. The features of the texture fetching functions must be implemented in a software library. The host and device memory spaces must generally be kept distinct, implying that API functions copying between host and device memory spaces should still operate as specified. Removing this overhead is a potential target for future work.

Local memory regions must be allocated per thread. The simplest method accomplishing this is to change each local memory object into an array of objects accessed by the CUDA thread index. The shared memory regions, private to a thread block, should be dynamically allocated for the thread blocks actively executing. For shared memory arrays of fixed size, this can be done using the program function stack. However, CUDA allows shared array of statically unspecified size, determined at kernel launch time. In C, this is most feasibly addressed by dynamically allocating a shared memory buffer of the appropriate size for each actively executing thread block. This is addressed in the runtime portion of the system.

The runtime environment is responsible for the execution of the programming model, given the adapted kernel functions generated by the compiler. Considering the thread blocks as work units, the runtime essentially implements a bulk-synchronous parallism model. It is responsible for the parallel processing of the work units within a grid, ensuring that different grids will be synchronized with each other and with the host.

5. Structured Microthreading

Consider a common case in which a kernel function has no synchronization. In this case, complex microthreading techniques are unnecessary, as the threads can be interleaved in any way we desire, including complete serialization. When barrier synchronization is present, complete serialization is not possible, but unstructured control flow caused by the added *goto* statements to and from the restart points of the previous approach is less easily analyzed by most compilers, especially for optimizations like automatic vectorization. The improved approach described in this section summarizes a variation of previous work [23] taking advantage of the synchronization restrictions to more efficiently implement microthreading.

Algorithm 1 partitions an SPMD program with textually-aligned static barriers and regular control flow into groups of statements not containing barrier synchronization. For each statement in sequence, we examine whether it is or contains a barrier statement. If not, it is included in the current partition. If it is a barrier statement, it defines a partition boundary, ending the current partition and beginning another. If it is a control-flow construct containing a barrier, then by the restrictions on the correct usage of barriers, all

```
Input: List of Statements $F$ in AST representation
Output: List $X$ of Code Partitions Free of Barriers
Begin new partition $P$;
while $F$ has next statement $S$ do
    switch type of statement $S$ do
        case barrier
            Add $P$ to $X$;
            $P$ = new partition;
        end
        case simple_statement
            Add $S$ to $P$;
        end
        case seq
            Prepend statements comprising $S$ to $F$;
        end
        otherwise
            if $S$ contains a barrier statement then
                Add $P$ to $X$;
                Invoke algorithm recursively on the body
                of $S$, producing a list $L$ of partitions
                within $S$; Append $L$ to $X$;
                $P$ = new partition;
            else
                Add $S$ to $P$;
            end
        end
    end
end
if $P$ not empty then
    Add $P$ to $X$;
end
```

Algorithm 1: Construction of code partitions free of barriers

```
1   __global__ small_mm_list(float* A_list, float* B_list,
                              , const int size)
    {
2     float sum[];
3     int matrix_start, col[], row[], out_index[], i[];
      for( each tid ) {

4       matrix_start = blockIdx.x * size * size;
5       col[tid] = matrix_start + tid.x;
6       row[tid] = matrix_start + (tid.y * size);

7       sum[tid] = 0.0;

8       for(i[tid] = 0; i[tid] < size; i[tid]++)
9         sum[tid] += A_list[row[tid] + i[tid]] *
                      B_list[col[tid] + (i[tid]*size)];
      }
10
      for( each tid ) {
11      out_index[tid] = matrix_start +
                          (tid.y * size) + tid.x;
12      A_list[out_index[tid]] = sum[tid];
      }
    }
```

Figure 4: Partitioned translation of our example kernel

threads must reach or not reach the construct, making it a valid partition boundary itself. The same algorithm is invoked recursively on the internal contents of the construct to partition the statements within.

These partitions define regions of code where the execution of different CUDA threads may be interleaved in any way, including complete serialization, as shown in Figure 4, where each partition is enclosed within a nested loop structure iterating through all thread indexes. Comparing Figure 4 to the previous Figure 3, we see that both perform the same sequential ordering of the original statements. However, Figure 4 does so with significantly less complex code in comparison, both inherently simpler and more easily analyzable for later optimization. For each statement of the program, the code generator also finds references to variables in the local memory space in that statement, and conservatively converts these into references to the replicated arrays.

6. Optimizations Enabled

Programmers writing parallel software make significant tradeoffs between the cost of redundant computation among parallel execution units and the cost of synchronization and communication. However, when these parallel applications are serialized to execute on a sequential processor, the cost of communication largely vanishes, and redundant computation often no longer makes sense. In sequential-program compilers, redundancy removal has been very successful, but somewhat limited by the conservative assumptions necessary to preserve sequential semantics when analysis falls short. However, when the sequential program is actually an explicitly parallel program serialized, the need for analysis is either greatly reduced or removed entirely, as interthread ordering semantics are much more loosely constrained than a typical sequential loop nest. While such optimizations should be possible within the baseline approach, it would not be possible to leverage the existing work on loop nest transformations in that context.

Variance Analysis Opportunities for redundancy removal are exposed by discovering what portions of the kernel code will produce the same value for all thread indexes. Computation that was previously performed redundantly by multiple CUDA threads now can be executed once in the single CPU thread. The core of variance analysis is the forward program slice of each element of the thread index tuple. We compute these program slices, annotating each statement with those program slices they comprise. We refer to these annotations as variance vectors. For instance, statement 9 of our example kernel has a variance vector of `(x,y)`, because it depends on the results of statements 5 and 6 that respectively read the `x` and `y` index components. Implicitly, atomic intrinsics are considered as a use of each element of the thread index, as their return value could vary for each CUDA thread.

When no statement in a partition contains a particular element in its variance vector, the partition does not need to be executed for each value in the index range of that element. Its results are independent of that element of the conceptual thread index. In the simplest case, and perhaps the most common, a programmer could intend to only use a subset of the elements of the thread index tuple to distinguish threads, implicitly assuming that all of the other elements will have a constant value of 1. In this case, the programmer writes a kernel never using some elements of the thread index tuple. The variance analysis will not annotate any statement with an unused component, directing the code generator to not create any loops over those elements of the thread index for any partition. This is the case for our example kernel, where the *z* index is unused.

Adaptive Loop Nesting Even when loops over certain elements are required for a partition, perhaps not all statements in a partition require execution for all thread indexes, analogous to loop invariant removal. However, we propose a technique called *adaptive loop nesting* that is more general in that it simultaneously evaluates transformations equivalent

```
1  __global__ mm_list(float* A_list, float* B_list,
                              , const int size)
   {
2    float sum[];
3    int matrix_start, col[], row[], out_index, i;

4    matrix_start = blockIdx.x * size * size;
     for(tid.x = 0; tid.x < blockDim.x; tid.x++) {
5      col[tid] = matrix_start + tid.x;

       for(tid.y = 0; tid.y < blockDim.y; tid.y++) {
6        row[tid] = matrix_start + (tid.y * size);
7        sum[tid] = 0.0;

8        for(i = 0; i < size; i++)
9          sum[tid] += A_list[row[tid] + i] *
                          B_list[col[tid] + (i*size)];
       }
     }
10
     for(tid.x = 0; tid.x < blockDim.x; tid.x++)
       for(tid.y = 0; tid.y < blockDim.y; tid.y++) {
11       out_index = matrix_start +
                              (tid.y * size) + tid.x;
12       A_list[out_index] = sum[tid];
     }
   }
```

Figure 5: Optimized translation of our example kernel

to loop interchange, loop fission, and loop invariant removal to achieve the best redundancy removal, similar to polyhedral modeling of loop nests for sequential languages [8]. The significant distinction from typical loop-nest optimization is that all iterations can be assumed independent without analysis because of their origin from parallel threads.

The compiler may generate loops over thread index elements only around those statements that contain that element in their variance vector. To remove loop overhead, the compiler may fuse adjacent statement groups where one has a variance vector that is a subset of the other. All of the traditional cost analysis applied to loop fusion operations may apply here.

Typical cost analysis must be used to determine cases such as statements 5-9 of our example kernel. Statements 7-9 must be included in a loop nest over both x and y components of the conceptual thread index, as the computation is unique to each CUDA thread. As each of statements 5 and 6 is only dependent on one index element, either can be merged into a loop nest with statements 7-9, inside the outer loop over one component but before the inner loop of the other index. However, choosing either statement 5 or 6 to merge will lead to one of two choices for the other. We may choose to force the other into the innermost loop, causing unnecessary redundant execution, since it was independent of one of the loops now containing it. Otherwise, me must enclose it in an extra, separate loop nest for that statement alone, incurring extra control overhead. We chose a cost heuristic that in this case would determine that the extra control overhead is more costly, and would generate the control flow observed in Figure 5 that redundantly executes statement 6 for every x index.

Optimizing Local Variable Replication We note that because of the serialization of the computation in the fine-grained threads, not all data conceptually private to each thread must necessarily be instantiated as separate memory locations per thread. In particular, it is not necessary to create private memory locations for values that have a live range completely contained within a partition. In such cases, one memory location reused by all threads is

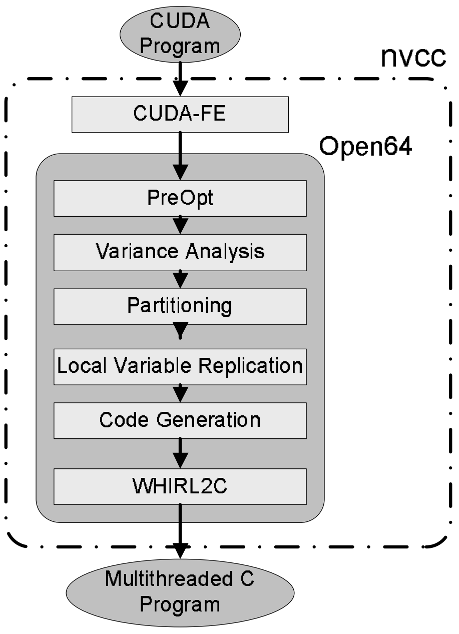

Figure 6: Compiler implementation diagram

sufficient. Another case is where, even though the variable is live through multiple partitions, its value is thread-invariant. This is the case when a variable definition has an empty variance vector.

Two cases arise in which variable replication must be applied to the output value of an assignment with a non-empty variance vector. The first is if a value defined by the assignment reaches a use in another partition. As stated previous, values with a live range completely contained within a partition will never need to be saved for the same conceptual thread to use in some later partition. The second is if, in the presence of the loop over thread indexes placed around the partition, the defined value would reach a use that it previously would not have.

Assignments with an empty variance vector technically never need to write to a replicated location, such as statement 4 of Figure 5. However, we decided that for any use reachable by at least one replicated definition, all its potential definitions must write to the replicated location for simplicity.

Minimal variable replication and adaptive loop nesting share an interesting interplay in that the maximal fusing of loops over indexes can introduce additional cases requiring replication. This has been well established in work on loop fusion. The final results of these optimization algorithms would result in a generated kernel code like that shown in Figure 5.

7. Implementation

The compiler is implemented within NVIDIA's production CUDA compilation toolchain. The toolchain provides a CUDA compiler driver, called nvcc. We added a new compiler flag enabling multicore compilation. The compiler structure is shown in Figure 6. At a high level the compiler consists of two main components: a frontend (CUDAFE) and the Open64 [9] high-level backend. CUDAFE is the standard CUDA production compiler front-end without modifications, just as it is used for GPU compilation.

In our implementation we generate HI-WHIRL intermediate representation (IR) for the Open64 backend infrastructure [9]. We implemented all the optimizing transformations at the HI-WHIRL level, chosen because almost all machine-independent analysis and optimization passes are available there [6]. The backend consists of five main components.

PreOpt- We use the standard Open64 optimizer to perform a few simple optimizations and, more importantly, to generate data flow information in the form of def-use chains.

Variance Analysis- The variance analysis we described earlier computes forward program slices on the thread index variables, annotating every statement with the components of the `threadIdx` variable on which that statement depends.

Partitioning- The partitioning algorithm described in Section 5 builds a list of partitions and, within each partition, collects a list of statements.

Local Variable Replication- Def-use chains restricted to the set of local variables of a function determine which variable references are read and written in multiple partitions. Each statement is annotated with the list of variable references within that statement needing to reference the expanded version of the variable.

Code Generation- This phase completes the generation of IR that is the complete, optimized transformation of the input into executable code. It traverses each partition, grouping adjacent statements if desirable given their varience vectors. It also transforms statements to use replicated versions of variables as necessary. Finally, it surrounds each grouped cluster of statements within a partition by the necessary thread loops, as required by the variance vectors of those statements.

WHIRL2C- We use the WHIRL2C [5] component from the Open64 distribution to generate C code from the transformed IR.

Thread blocks in the CUDA programming model represent independent tasks, each embodied by a sequential program following our compiler's translation. Many frameworks exist for distributing such parallel tasks to processors. Our implementation uses POSIX threads as an example. The runtime system creates several OS worker threads, the number of which can be controlled by an environment variable. At a kernel launch, the number of CUDA thread blocks in the grid to be launched is statically partitioned to the runtime threads. Each runtime thread executes its chunk sequentially and waits on a barrier. When all runtime threads reach the barrier, the grid has completed, and control is returned to the host thread.

8. Performance Evaluation

We present results on the eight CUDA benchmarks in Table 2 from application fields including fluid dynamics, astrophysics, and financial modeling. These applications were written specifically for a GPU target architecture, and have shown significant performance on that platform, some reported in previous work [18]. For benchmarking, we used an Intel Core2 Quad processor system running RedHat Enterprise Linux 4 (Update 7). We use gcc version 3.4.6 as the final C compiler, with -O3 optimization for all tests.

Table 3 shows that optimizations of the structured microthreading implementation dramatically reduced the number of replicated variables, with direct effect on reducing cache pressure. The number of references to replicated variables is also consequently reduced, intuitively leading gcc to promote a larger fraction of variable accesses to register accesses. The variance analysis correctly detected that, out of all of the benchmarks, only `tpacf` used two dimensions of the thread index, while all the other applications used only one.

Benchmark	App. domain	Kernel lines	Static barriers
petrinet	stochastic models	191	5
blinn	volume rendering	155	0
blackscholes	financial models	43	0
nbody	astrophysics sim.	180	3
lbm	fluid sim.	285	1
tpacf	astronomy data processing	98	4
binoption	financial models	121	5
FDTD	electromagnetic simulation	263	6

Table 2: Benchmark summary

Benchmark	Local objects	Static local object references	Replicated local objects	Static references to replicated objects
petrinet	72	623	0	0
blinn	93	343	0	0
blackscholes	35	133	0	0
nbody	82	498	18	141
lbm	110	1269	11	51
tpacf	36	196	6	25
binoption	51	215	6	6
FDTD	46	481	13	94

Table 3: Static Results of Optimizing Transformations

Figure 7 shows the benefits of our optimizations over a traditional microthreaded approach. Those applications with the least performance differences, `blinn` and `blackscholes`, do not use any synchronization within the CUDA kernel. In these cases, the performance benefits of the structured implementation are primarily due to the removal of the redundant local memory objects, as the control flow structure is practically the same between the two implementations. The rest of the applications do use synchronization, and gain significant performance benefits from the structured implementation, with an average of approximately 2× performance difference between the baseline and structured implementations of microthreading.

The most extreme cases of disparity between structured and unstructured microthreading were `BinOption` and `FDTD`. These were also the applications with the most synchronization, showing that the advantage of strutured microthreading and optimization generally increases with kernel program complexity.

Finally, we can see that the performance compared to a native C application varies widely. This is to be expected, as the implementation decisions were made in different programming models, although the task and general algorithm were fixed. `petrinet` and `FDTD` required the most parrallel algorithm implementation overhead, reflected in the comparison with sequential execution. Some applications even saw single thread performance gains over the existing C implementation. This indicates that the optimization effort spent

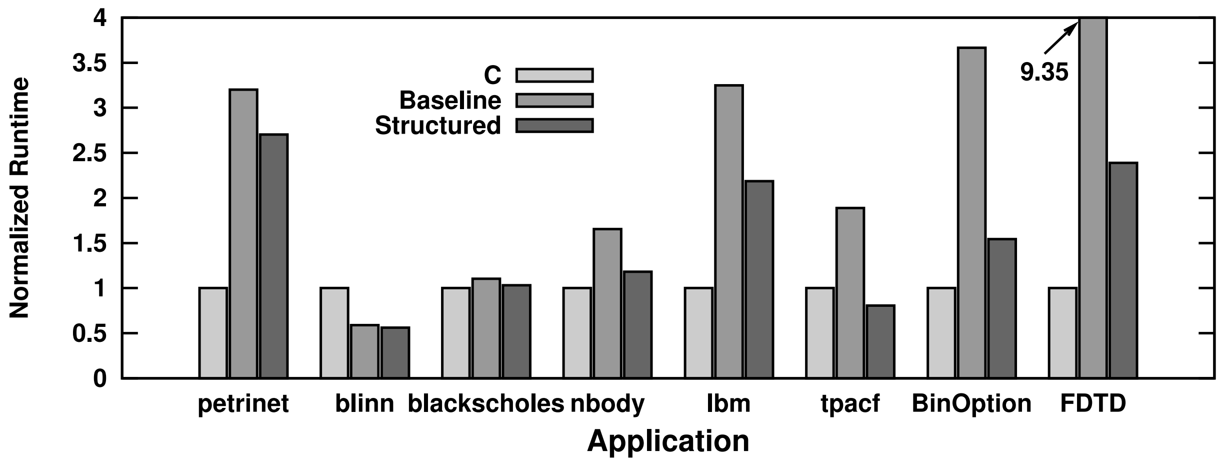

Figure 7: Translated CUDA application runtime relative to a native C implementation, each using one CPU execution thread. Only nominal programmer optimization effort was applied to either the C or CUDA versions of the code.

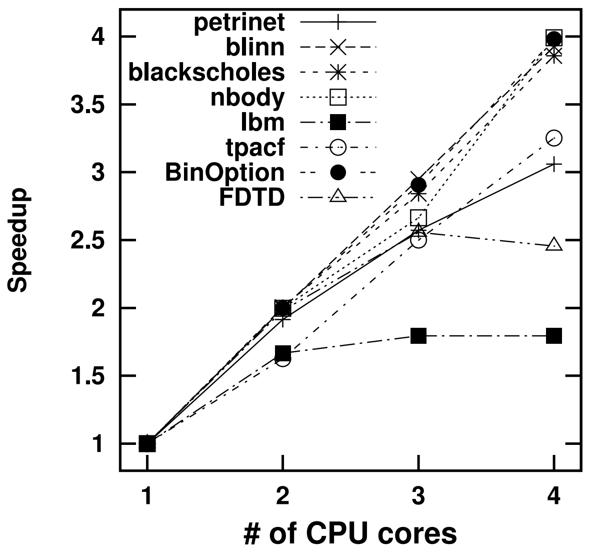

Figure 8: Application scaling from 1 to 4 threads.

on the CUDA implementation, for the GPU, was more effective for the CPU than the optimization effort spent on the C implementation.

All applications also saw significant performance gains from multithreading across the coarse-grained cores. We can see in Figure 8 that the performance scaling of the translated applications is very good, with close to ideal linear scaling for a small number of processor cores for most applications. The only application that reaches a scaling ceiling on our test system is `lbm`, as the application becomes bottlenecked by system memory bandwidth. Several other applications show somewhat less than ideal scaling, primarily due to load imbalance caused by our simplistic work partitioning implementation developed under the assumption of large numbers of equal-latency tasks. The two applications most affected by load imbalance are `tpacf` and `petrinet`, which have large variations in the runtimes of each block. A large existing body of work explores more effective dynamic work scheduling policies [15] applicable to our implementation would likely move some of the applications closer to the ideal scaling curve.

9. Conclusions

We have described techniques for efficiently implementing the CUDA programming model on a conventional multiprocessor CPU architecture. We have described a baseline microthreading approach, showing that a microthreading approach based on structured control flow has significant comparative performance advantages, in part due to additional optimizations that are enabled.

We observe that a fine-grained SPMD decomposition can be translated into more coarse-grained work units effectively, but only with reasonable restrictions on the synchronization model. Fine-grained threads that may interact arbitrarily must resort to some form of unstructured microthreading, which has shown to as much as double execution times compared to the structured approach, and in no case was it better. Our results also suggest that there is a class of parallel kernels where the finely-threaded version of the code shows parity with a native C implementation in single-thread performance.

Finally, our results have shown a particular software engineering advantage for current CUDA developers requiring some CPU fallback implementation when CUDA is not installed on a particular client's system. Using these techniques, such developers could translate their CUDA code directly into multithreaded C that is almost always better than a quickly written sequential program on a small multiprocessor typical in today's systems, while still keeping a single code base.

Acknowledgments

We would like the thank NVIDIA corporation for its support, and our formal and informal reviewers for their feedback.

References

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 342–354, 1998.

[2] H. A. Andrade and S. Kovner. Software synthesis from dataflow models for embedded software design in the G programming language and the LabVIEW development environment. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 1705–1709, 1998.

[3] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen. Efficient, portable implementation of asynchronous multi-place programs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 271–282, New York, NY, USA, 2009. ACM.

[4] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*, pages 104–111, June 2008.

[5] W.-Y. Chen. Building a source-to-source UPC-to-C translator. Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, 2004.

[6] F. Chow, S. Chan, R. Kennedy, S. ming Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on ssa form. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 273–286, 1997.

[7] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, 1(1):42–50, 1999.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20:23–53, 1991.

[9] G. Gao, J. Amaral, J. Dehnert, and R. Towle. The SGI Pro64 compiler infrastructure. Tutorial. October 2000.

[10] Gregory Diamos. The design and implementation Ocelot's dynamic binary translator from PTX to multi-core x86. Technical Report GIT-CERCS-09-18, Georgia Institute of Technology, 2009.

[11] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[12] Khronos OpenCL Working Group. The OpenCL Specification, May 2009.

[13] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of 14th ACM Symposium on Principles and Practice of Parallel Programming*, pages 101–110, 2008.

[14] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for Brook streaming applications on multiprocessors. In *Proceedings of the 4th International Symposium on Code Generation and Optimization*, pages 196–207, March 2006.

[15] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 104–113, July 1992.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[17] OpenMP Architecture Review Board. OpenMP application program interface, May 2005.

[18] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, February 2008.

[19] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8(1):474, 2007.

[20] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 181–192, New York, NY, USA, 2009. ACM.

[21] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, September 2007.

[22] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. W. Hwu, Z.-P. Liang, and B. P. Sutton. Accelerating advanced MRI reconstructions on GPUs. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 261–272, 2008.

[23] J. A. Stratton, S. S. Stone, and W. mei W. Hwu. MCUDA: An effective implementation of CUDA kernels for multi-core CPUs. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, pages 16–30, July 2008.

[24] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.