MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs

John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu

Center for Reliable and High-Performance Computing and
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{stratton,ssstone2,hwu}@crhc.uiuc.edu

Abstract. CUDA is a data parallel programming model that supports several key abstractions - thread blocks, hierarchical memory and barrier synchronization - for writing applications. This model has proven effective in programming GPUs. In this paper we describe a framework called MCUDA, which allows CUDA programs to be executed efficiently on shared memory, multi-core CPUs. Our framework consists of a set of source-level compiler transformations and a runtime system for parallel execution. Preserving program semantics, the compiler transforms threaded SPMD functions into explicit loops, performs fission to eliminate barrier synchronizations, and converts scalar references to thread-local data to replicated vector references. We describe an implementation of this framework and demonstrate performance approaching that achievable from manually parallelized and optimized C code. With these results, we argue that CUDA can be an effective data-parallel programming model for more than just GPU architectures.

1 Introduction

In February of 2007, NVIDIA released the CUDA programming model for use with their GPUs to make them available for general purpose application programming [1]. However, the adoption of the CUDA programming model has been limited to those programmers willing to write specialized code that only executes on certain GPU devices. This is undesirable, as programmers who have invested the effort to write a general-purpose application in a data-parallel programming language for a GPU should not have to make an entirely separate programming effort to effectively parallelize the application across multiple CPU cores.

One might argue that CUDA's exposure of specialized GPU features limits the efficient execution of CUDA kernels to GPUs. For example, in a typical usage case of the CUDA programming model, programmers specify hundreds to thousands of small, simultaneously active threads to achieve full utilization of GPU execution resources. However, a current CPU architecture currently supports only up to tens of active thread contexts. On the other hand, some language features in the CUDA model can be beneficial to performance on a CPU, because these features encourage the programmer to use more disciplined

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 16–30, 2008.

control flow and expose data locality. Section 2 describes in more detail the key CUDA language features and a deeper assessment of why we expect many CUDA features to map well to a CPU architecture for execution. We propose that if an effective mapping of the CUDA programming model to a CPU architecture is feasible, it would entail translations applied to a CUDA program such that the limiting features of the programming model are removed or mitigated, while the beneficial features remain exposed when possible.

Section 3 describes how the MCUDA system translates a CUDA program into an efficient parallel C program. Groups of individual CUDA threads are collected into a single CPU thread while still obeying the scheduling restrictions of barrier synchronization points within the CUDA program. The data locality and regular control encouraged by the CUDA programming model are maintained through the translation, making the resulting C program well suited for a CPU architecture.

The implementation and experimental evaluation of the MCUDA system is presented in Section 4. Our experiments show that optimized CUDA kernels utilizing MCUDA achieve near-perfect scaling with the number of CPU cores, and performance comparable to hand-optimized multithreaded C programs. We conclude this paper with a discussion of related work in Section 5 and some closing observations in Section 6.

2 Programming Model Background

On the surface, most features included in the CUDA programming model seem relevant only to a specific GPU architecture. The primary parallel construct is a data-parallel, SPMD *kernel* function. A kernel function invocation explicitly creates many CUDA threads (hereafter referred to as *logical threads.*) The threads are organized into multidimensional arrays that can synchronize and quickly share data, called thread *blocks*. These thread blocks are further grouped into another multidimensional array called a *grid*. Logical threads within a block are distinguished by an implicitly defined variable *threadIdx*, while blocks within a grid are similarly distinguished by the implicit variable *blockIdx*. At a kernel invocation, the programmer uses language extensions to specify runtime values for each dimension of threads in a thread block and each dimension of thread blocks in the grid, accessible within the kernel function through the variables *blockDim* and *gridDim* respectively. In the GPU architecture, these independent thread blocks are dynamically assigned to parallel processing units, where the logical threads are instantiated by hardware threading mechanisms and executed.

Logical threads within CUDA thread blocks may have fine-grained execution ordering constraints imposed by the programmer through barrier synchronization intrinsics. Frequent fine-grained synchronization and data sharing between potentially hundreds of threads is a pattern in which CPU architectures typically do not achieve good performance. However, the CUDA programming model does restrict barrier synchronization to within thread blocks, while different thread blocks can be executed in parallel without ordering constraints.

The CUDA model also includes explicitly differentiated memory spaces to take advantage of specialized hardware memory resources, a significant departure from the unified memory space of CPUs. The *constant* memory space uses a small cache of a few kilobytes optimized for high temporal locality and accesses by large numbers of threads across multiple thread blocks. The *shared* memory space maps to the scratchpad memory of the GPU, and is shared among threads in a thread block. The *texture* memory space uses the GPU's texture caching and filtering capabilities, and is best utilized with data access patterns exhibiting 2-D locality. More detailed information about GPU architecture and how features of the CUDA model affect application performance is presented in [2].

In the CUDA model, logical threads within a thread block can have independent control flow through the program. However, the NVIDIA G80 GPU architecture executes logical threads in SIMD bundles called *warps*, while allowing for divergence of thread execution using a stack-based reconvergence algorithm with masked execution [3]. Therefore, logical threads with highly irregular control flow execute with greatly reduced efficiency compared to a warp of logical threads with identical control flow. CUDA programmers are strongly encouraged to adopt algorithms that force logical threads within a thread block to have very similar, if not exactly equivalent, execution traces to effectively use the implicitly SIMD hardware effectively. In addition, the CUDA model encourages data locality and reuse for good performance on the GPU. Accesses to the global memory space incur uniformly high latency, encouraging the programmer to use regular, localized accesses through the scratchpad shared memory or the cached constant and texture memory spaces.

A closer viewing of the CUDA programming model suggests that there could also be an efficient mapping of the execution specified onto a current multicore CPU architecture. At the largest granularity of parallelism within a kernel, blocks can execute completely independently. Thus, if all logical threads within a block occupy the same CPU core, there is no need for inter-core synchronization during the execution of blocks. Thread blocks often have very regular control flow patterns among constituent logical threads, making them amenable to the SIMD instructions common in current x86 processors [4,5]. In addition, thread blocks often have the most frequently referenced data specifically stored in a set of thread-local or block-shared memory locations, which are sized such that they approximately fit within a CPU core's L1 data cache. Shared data for the entire kernel is often placed in constant memory with a size limit appropriate for an L2 cache, which is frequently shared among cores in CPU architectures. If a translation can be designed such that these attributes are maintained, it should be possible to generate effective multithreaded CPU code from the CUDA specification of a program.

3 Kernel Translation

While the features of the model seem promising, the mapping of the computation is not straightforward. The conceptually easiest implementation is to spawn an

OS thread for every GPU thread specified in the programming model. However, allowing logical threads within a block to execute on any available CPU core mitigates the locality benefits noted in the previous section, and incurs a large amount of scheduling overhead. Therefore, we propose a method of translating the CUDA program such that the mapping of programming constructs maintains the locality expressed in the programming model with existing operating system and hardware features.

There are several challenging goals in effectively translating CUDA applications. First, each thread block should be scheduled to a single core for locality, yet maintain the ordering semantics imposed by potential barrier synchronization points. Without modifying the operating system or architecture, this means the compiler must somehow manage the execution of logical threads in the code explicitly. Second, the SIMD-like nature of the logical threads in many applications should be clearly exposed to the compiler. However, this goal is in conflict with supporting arbitrary control flow among logical threads. Finally, in a typical load-store architecture, private storage space for every thread requires extra instructions to move data in and out of the register file. Reducing this overhead requires identifying storage that can be safely reused for each thread.

The translation component of MCUDA which addresses these goals is composed of three transformation stages: iterative wrapping, synchronization enforcement, and data buffering. For purposes of clarity, we consider only the case of a single kernel function with no function calls to other procedures, possibly through exhaustive inlining. It is possible to extend the framework to handle function calls with an interprocedural analysis, but this is left for future work. In addition, without loss of generality, we assume that the code does not contain *goto* or *switch* statements, possibly through prior transformation [6]. All transformations presented in this paper are performed on the program's abstract syntax tree (AST).

3.1 Transforming a Thread Block into a Serial Function

The first step in the transformation changes the nature of the kernel function from a per-thread code specification to a per-block code specification, temporarily ignoring any potential synchronization between threads. Figure 1 shows an example kernel function before and after this transformation. Execution of logical threads is serialized using nested loops around the body of the kernel function to execute each thread in turn. The loops enumerate the values of the previously implicit threadIdx variable and perform a logical thread's execution of the enclosed statements on each iteration. For the remainder of the paper, we will consider this introduced iterative structure a *thread loop*. Local variables are reused on each iteration, since only a single logical thread is active at any time. Shared variables still exist and persist across loop iterations, visible to all logical threads. The other implicit variables must be provided to the function at runtime, and are therefore added to the parameter list of the function.

By introducing a thread loop around a set of statements, we are making several explicit assumptions about that set of statements. The first is that the

```
void cengery(numatoms, gridspacing, energygrid[] )
{
        int x - blockIdx.x * blockDim.x
                        + threadIdx.x;
        int y - blockIdx.y * blockDim.y
                        + threadIdx.y;
        int outIdx - gridDim.x * blockDim.x * y
                        + x;

        float energy - 0.0;
        int atomid-0;
        while(atomid<numatoms) {
             …
        }
        energygrid[outIdx] - energy;
}
```

```
void cengery(numatoms, gridspacing, energygrid[],
               blockDim, blockIdx, gridDim)
{
    dim3 threadIdx;
    // Thread Loop
    for(threadIdx.y = 0;
          threadIdx.y < blockDim.y;
          threadIdx.y++)
      for(threadIdx.x = 0;
            threadIdx.x < blockDim.x;
            threadIdx.x++)
    {
        int x - blockIdx.x * blockDim.x
                        + threadIdx.x;
        int y - blockIdx.y * blockDim.y
                        + threadIdx.y;
        int outIdx - gridDim.x*blockDim.x * y
                        + x;

        float energy - 0.0;
        int atomid-0;
        while(atomid < numatoms) {
             …
        }
        energygrid[outIdx] - energy;
    }
    // end Thread Loop;
}
```

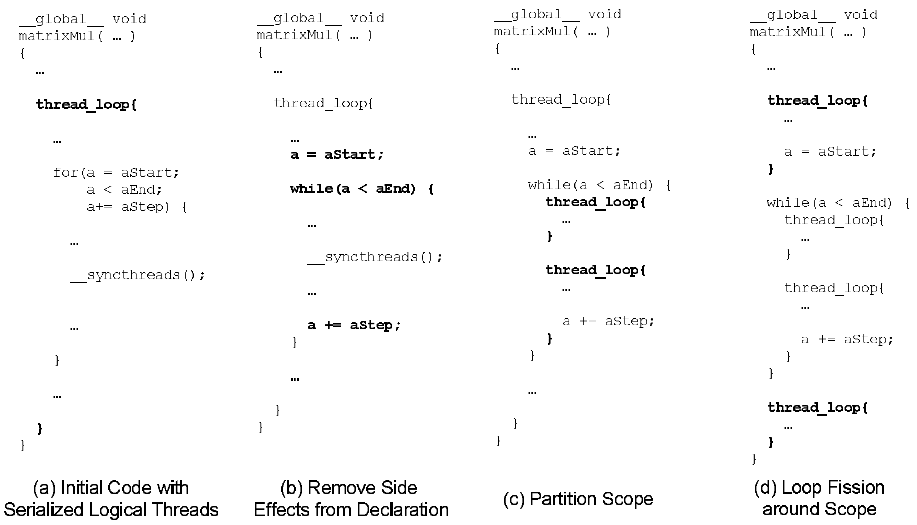
Fig. 1. Introducing a thread loop to serialize logical threads in Coulombic Potential

program allows each logical thread to execute those statements without any synchronization between threads. The second is that there can be no side entries into or side exits out of the thread loop body. If the programmer has not specified any synchronization point and the function contains no explicit return statement, no further transformation is required, as a function cannot have side entry points, and full inlining has removed all side-exits. In the more general case, where using a single thread loop is insufficient for maintaining program semantics, we must partition the function into sets of statements which do satisfy these properties.

3.2 Enforcing Synchronization with Deep Fission

A thread loop implicitly introduces a barrier synchronization among logical threads at its boundaries. Each logical thread executes to the end of the thread loop, and then "suspends" until every other logical thread (iteration) completes the same set of statements. Therefore, a loop fission operation essentially partitions the statements of a thread loop into two sets of statements with an implicit barrier synchronization between them. A synchronization point found in the immediate scope of a thread loop can be thus enforced by applying a loop fission operation at the point of synchronization.

Although a loop fission operation applied to the thread loop enforces a barrier synchronization at that point, this operation can only be applied at the scope of the thread loop. As mentioned before, the general case requires a transformation that partitions statements into thread loops such that each thread loop contains no synchronization point, and each thread loop boundary is a valid synchronization point. For example, consider the case of Figure 2. There are at minimum four groups of statements required to satisfy the requirements for thread loops: one leading up to the for loop (including the loop initialization



```
__global__ void
matrixMul( … )
{
  …

  thread_loop{

    …

    for(a = aStart;
        a < aEnd;
        a+= aStep) {

      …

      __syncthreads();

      …

    }

    …

  }
}
```

(a) Initial Code with Serialized Logical Threads

```
__global__ void
matrixMul( … )
{
  …

  thread_loop{

    …
    a = aStart;

    while(a < aEnd) {

      …

      __syncthreads();

      …

      a += aStep;
    }

    …

  }
}
```

(b) Remove Side Effects from Declaration

```
__global__ void
matrixMul( … )
{
  …

  thread_loop{

    …
    a = aStart;

    while(a < aEnd) {
      thread_loop{
        …
      }

      thread_loop{
        …

        a += aStep;
      }
    }

    …

  }
}
```

(c) Partition Scope

```
__global__ void
matrixMul( … )
{
  …

  thread_loop{
    …

    a = aStart;
  }

  while(a < aEnd) {
    thread_loop{
      …
    }

    thread_loop{
      …

      a += aStep;
    }
  }

  thread_loop{
    …
  }
}
```

(d) Loop Fission around Scope

Fig. 2. Applying deep fission in Matrix Multiplication to enforce synchronization

statement), one for the part of the loop before the synchronization point, one after the synchronization point within the loop (including the loop update), and finally the trailing statements after the loop.

In this new set of thread loops, the logical threads will implicitly synchronize every time the loop conditional is evaluated, in addition to the programmer-specified synchronization point. This is a valid transformation because of the CUDA programming model's requirement that control flow affecting a synchronization point must be thread-independent. This means that if the execution of a synchronization point is control-dependent on a condition, that condition must be thread-invariant. Therefore, if any thread arrives at the conditional evaluation, all threads must reach that evaluation, and furthermore must evaluate the conditional in the same way. Such a conditional can be evaluated outside of a thread loop once as a representative for all logical threads. In addition, it is valid to force all threads to synchronize at the point of evaluation, and thus safe to have thread loops bordering and comprising the body of the control structure.

In describing our algorithm for enforcing synchronization points, we first assume that all control structures have no side effects in their declarations. We enforce that *for* loops must be transformed into *while* loops in the AST, removing the initialization and update expressions. In addition, all conditional evaluations with side effects must be removed from the control structure's declaration, and assigned to a temporary variable, which then replaces the original condition in the control structure. Then, for each synchronization statement S, we apply Algorithm 1 to the AST with S as the input parameter.

After this algorithm has been applied with each of the programmer-specified synchronization points as input, the code may still have some control flow for which the algorithm has not properly accounted. Recall that thread loops assume

Algorithm 1. Deep Fission around a Synchronization Statement S

```
loop
  if Immediate scope containing $S$ is not a thread loop then
    Partition all statements within the scope containing $S$ into thread loops. Statements before and after $S$ form two thread loops. In the case of an if-else construct, this also means all statements within the side of the construct not containing $S$ are formed into an additional thread loop. {See Figure 2(c)}
  else
    Apply a loop fission operation to the thread loop around $S$ and return {(See Figure 2(d).)}
  end if
  $S \leftarrow$ Construct immediately containing $S$ {Parent of $S$ in the AST}
end loop
```

that there are no side entries or side exits within the thread loop body. Control flow statements such as *continue*, *break*, or *return* may not be handled correctly when the target of the control flow is not also within the thread loop. Figure 3(b) shows a case where irregular control flow would result in incorrect execution. In some iterations of the outer loop, all logical threads may avoid the break and synchronize correctly. In another iteration, all logical threads may take the break, avoiding synchronization. However, in the second case, control flow would leave the first thread loop before all logical threads had finished the first thread loop, inconsistent with the program's specification. Again, we note that since the synchronization point is control-dependent on the execution of the break statement, the break statement itself can be a valid synchronization point according to the programming model.

Therefore, the compiler must pass through the AST at least once more to identify these violating control flow statements. At the identification of a control flow statement S whose target is outside its containing thread loop, Algorithm 1 is once again applied, treating S as a synchronization statement. For the example of Figure 3, this results in the code shown in Figure 3(c). Since these

```
thread_loop{
  while() {

    ...
    if()
      break;
    ...

    syncthreads();

    ...

  }
}
```

(a) Initial Code with Serialized Logical Threads

```
while() {
  thread_loop{
    ...
    if()
      break;
    ...
  }
  \\syncthreads();
  thread_loop{
    ...
  }
}
```

(b) Synchronized at Barrier Function

```
while() {
  thread_loop{
    ...
  }
  if()
    break;
  thread_loop{
    ...
  }
  \\syncthreads();
  thread_loop{
    ...
  }
}
```

(c) Synchronized at Control Flow Point

Fig. 3. Addressing unstructured control flow. The break statement is treated as an additional synchronization statement for correctness.

transformations more finely divide thread loops, they could reveal additional control flow structures that violate the thread loop properties. Therefore, this irregular control flow identification and synchronization step is applied iteratively until no additional violating control flow is identified.

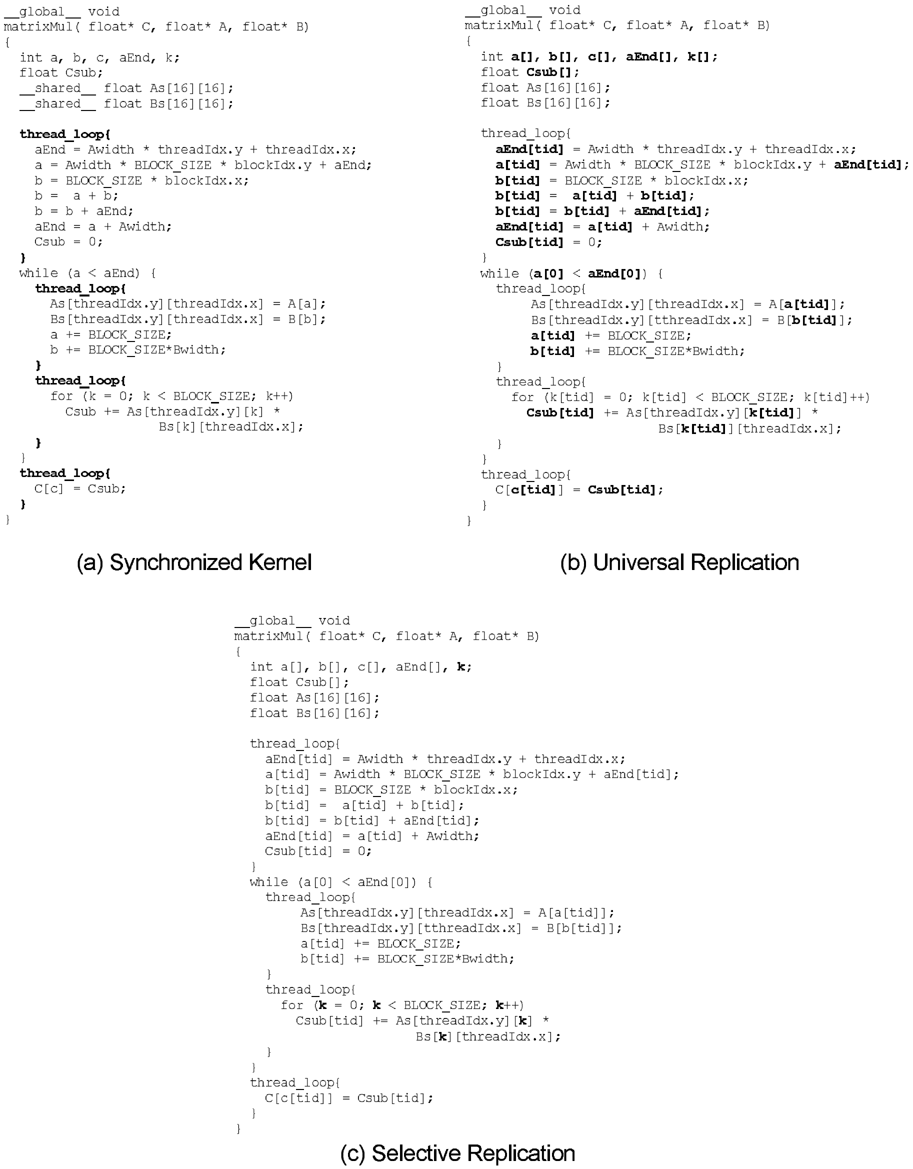
The key insight is that we are not supporting arbitrary control flow among logical threads within a block, but leveraging the restrictions in the CUDA language to define a single-threaded ordering of the instructions of multiple threads which satisfies the partial ordering enforced by the synchronization points. This "over-synchronizing" allows us to completely implement a "threaded" control flow using only iterative constructs within the code itself. The explicit synchronization primitives may now be removed from the code, as they are guaranteed to be bounded by thread loops on either side, and contain no other computation. Because only barrier synchronization primitives are provided in the CUDA programming model, no further control-flow transformations to the kernel function are needed to ensure proper ordering of logical threads. Figure 4(a) shows the matrix multiplication kernel after this hierarchical synchronization procedure has been applied.

3.3 Replicating Thread-Local Data

Once the control flow has been restructured, the final task remaining is to buffer the declared variables as needed. Shared variables are declared once for the entire block, so their declarations simply need the *shared* keyword removed. However, each logical thread has a local store for variables, independent of all other logical threads. Because these logical threads no longer exist independently, the translated program must emulate private storage for logical threads within the block. The simplest implementation creates private storage for each thread's instance of the variable, analogous to scalar expansion [7]. This technique, which we call *universal replication*, fully emulates the local store of each logical thread by creating an array of values for each local variable, as shown in Figure 4(b). Statements within thread loops access these arrays by thread index to emulate the logical thread's local store.

However, universal replication is often unnecessary and inefficient. In functions with no synchronization, thread loops can completely serialize the execution of logical threads, reusing the same memory locations for local variables. Even in the presence of synchronization, some local variables may have live ranges completely contained within a thread loop. In this case, logical threads can still reuse the storage locations of those variables because a value of that variable is never referenced outside the thread loop in which it is defined. For example, in the case of Figure 4(b), the local variable k can be safely reused, because it is never live outside the third thread loop.

Therefore, to use less memory space, the MCUDA framework should only create arrays for local variables when necessary. A live-variable analysis determines which variables have a live value at the end of a thread loop, and creates arrays for those values only. This technique, called *selective replication*, results in the code shown in Figure 4(c), which allows all logical threads to use the same



```
__global__ void
matrixMul( float* C, float* A, float* B)
{
  int a, b, c, aEnd, k;
  float Csub;
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  thread_loop{
    aEnd = Awidth * threadIdx.y + threadIdx.x;
    a = Awidth * BLOCK_SIZE * blockIdx.y + aEnd;
    b = BLOCK_SIZE * blockIdx.x;
    b =  a + b;
    b = b + aEnd;
    aEnd = a + Awidth;
    Csub = 0;
  }
  while (a < aEnd) {
    thread_loop{
      As[threadIdx.y][threadIdx.x] = A[a];
      Bs[threadIdx.y][threadIdx.x] = B[b];
      a += BLOCK_SIZE;
      b += BLOCK_SIZE*Bwidth;
    }
    thread_loop{
      for (k = 0; k < BLOCK_SIZE; k++)
        Csub += As[threadIdx.y][k] *
                    Bs[k][threadIdx.x];
    }
  }
  thread_loop{
    C[c] = Csub;
  }
}
```

(a) Synchronized Kernel

```
__global__ void
matrixMul( float* C, float* A, float* B)
{
  int a[], b[], c[], aEnd[], k[];
  float Csub[];
  float As[16][16];
  float Bs[16][16];

  thread_loop{
    aEnd[tid] = Awidth * threadIdx.y + threadIdx.x;
    a[tid] = Awidth * BLOCK_SIZE * blockIdx.y + aEnd[tid];
    b[tid] = BLOCK_SIZE * blockIdx.x;
    b[tid] =  a[tid] + b[tid];
    b[tid] = b[tid] + aEnd[tid];
    aEnd[tid] = a[tid] + Awidth;
    Csub[tid] = 0;
  }
  while (a[0] < aEnd[0]) {
    thread_loop{
        As[threadIdx.y][threadIdx.x] = A[a[tid]];
        Bs[threadIdx.y][tthreadIdx.x] = B[b[tid]];
        a[tid] += BLOCK_SIZE;
        b[tid] += BLOCK_SIZE*Bwidth;
    }
    thread_loop{
      for (k[tid] = 0; k[tid] < BLOCK_SIZE; k[tid]++)
        Csub[tid] += As[threadIdx.y][k[tid]] *
                              Bs[k[tid]][threadIdx.x];
    }
  }
  thread_loop{
    C[c[tid]] = Csub[tid];
  }
}
```

(b) Universal Replication

```
__global__ void
matrixMul( float* C, float* A, float* B)
{
  int a[], b[], c[], aEnd[], k;
  float Csub[];
  float As[16][16];
  float Bs[16][16];

  thread_loop{
    aEnd[tid] = Awidth * threadIdx.y + threadIdx.x;
    a[tid] = Awidth * BLOCK_SIZE * blockIdx.y + aEnd[tid];
    b[tid] = BLOCK_SIZE * blockIdx.x;
    b[tid] =  a[tid] + b[tid];
    b[tid] = b[tid] + aEnd[tid];
    aEnd[tid] = a[tid] + Awidth;
    Csub[tid] = 0;
  }
  while (a[0] < aEnd[0]) {
    thread_loop{
        As[threadIdx.y][threadIdx.x] = A[a[tid]];
        Bs[threadIdx.y][tthreadIdx.x] = B[b[tid]];
        a[tid] += BLOCK_SIZE;
        b[tid] += BLOCK_SIZE*Bwidth;
    }
    thread_loop{
      for (k = 0; k < BLOCK_SIZE; k++)
        Csub[tid] += As[threadIdx.y][k] *
                          Bs[k][threadIdx.x];
    }
  }
  thread_loop{
    C[c[tid]] = Csub[tid];
  }
}
```

(c) Selective Replication

Fig. 4. Data replication in Matrix Multiplication

memory location for the local variable *k*. However, *a* and *b* are defined and used across thread loop boundaries, and must be stored into arrays.

References to a variable outside of the context of a thread loop can only exist in the conditional evaluations of control flow structures. Control structures must affect synchronization points to be outside a thread loop, and therefore must be uniform across the logical threads in the block. Since all logical threads should have the same logical value for conditional evaluation, we simply reference element zero as a representative, as exemplified by the while loop in Figure 4 (b-c).

It is useful to note that although CUDA defines separate memory spaces for the GPU architecture, all data resides in the same shared memory system in the MCUDA framework, including local variables. The primary purpose of the different memory spaces on the GPU is to specify access to the different caching mechanisms and the scratchpad memory. A typical CPU system provides a single, cached memory space, thus we map all CUDA memory types to this memory space.

3.4 Work Distribution and Runtime Framework

At this point in the translation process the kernels are now defined as block-level functions, and all that remains is, on kernel invocation, to iterate through the block indexes specified and call the transformed function once for every specified block index. For a CPU that gains no benefits from multithreading, this is an efficient way of executing the kernel computation. However, CPU architectures that do gain performance benefits from multithreading will likely not achieve full efficiency with this method. Since these blocks can execute independently according to the programming model, the set of block indexes may be partitioned arbitrarily among concurrently executing OS threads. This allows the kernels to exploit the full block-level parallelism expressed in the programming model.

4 Implementation and Performance Analysis

We have implemented the MCUDA automatic kernel translation framework under the Cetus source-to-source compilation framework [8], with slight modifications to the IR and preprocessor to accept ANSI C with the language extensions of CUDA version 0.8. MCUDA implements the algorithms presented in the previous section for kernel transformations and applies them to the AST intermediate representation of Cetus. The live variable analysis required for robust selective replication described in Section 3.3 is incomplete, but the current implementation achieves the same liveness results for all the applications presented in this section. For compatibility with the Intel C Compiler (ICC), we replace the CUDA runtime library with a layer interfacing to standard libc functions for memory management. We chose to implement the runtime assignment of blocks to OS threads with OpenMP, using a single “parallel for” pragma to express the parallelism. A large existing body of work explores scheduling policies of such loops in OpenMP and other frameworks [9,10,11,12], For our experiments, we use the default compiler implementation.

Figure 5 shows the kernel speedup of three applications: matrix multiplication of two 4kx4k element matrices (MM4K), Coulombic Potential (CP), and MRI-FHD, a computationally intensive part of high-resolution MRI image reconstruction. These applications have previously shown to have very efficient CUDA implementations on a GPU architecture [13]. The CPU baselines that

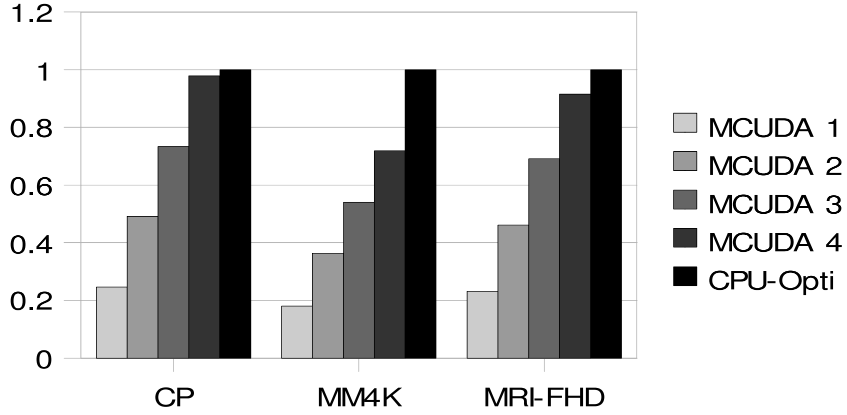


Fig. 5. Performance (inverse runtime) of MCUDA kernels relative to optimized CPU code. MCUDA results vary by the number of worker threads (1-4). CPU Opti implementations are parallelized across 4 threads.

we are measuring against are the most heavily optimized CPU implementations available to us, and are threaded by hand to make use of multiple CPU cores. All performance data was obtained on an Intel Core 2 Quad processor clocked at 2.66 GHz (CPU model Q6700). All benchmarks were compiled with ICC (version 10.1). Additionally, the CPU optimized matrix multiplication application uses the Intel MKL.

We can see that the performance scaling of this implementation is very good, with practically ideal linear scaling for a small number of processor cores. For each application, the performance of the CUDA code translated through the MCUDA framework is within 30% of the most optimized CPU implementation available. This suggests that the data tiling and locality expressed in effective CUDA kernels also gain significant performance benefits on CPUs, often replicating the results of hand-optimization for the CPU architecture. The regularly structured iterative loops of the algorithms were also preserved through the translation. The compiler vectorized the innermost loops of each application automatically, whether those were thread loops or loops already expressed in the algorithm.

Tuning CUDA kernels entails methodically varying a set of manual optimizations applied to a kernel. Parameters varied in this tuning process may include the number of logical threads in a block, unrolling factors for loops within the kernel, and tiling factors for data assigned to the scratchpad memory [14]. The performance numbers shown are the best results found by tuning each application. In the tuning process, we found that not all optimizations that benefit a GPU architecture are effective for compiling and executing on the CPU. In these applications, manual unrolling in the CUDA source code almost always reduced the effectiveness of the backend compiler, resulting in poorer performance. Optimizations that spill local variables to shared memory were also not particularly effective, since the shared memory and local variables reside in the same memory space on the CPU.

In general, the best optimization point for each application may be different depending on whether the kernel will execute on a GPU or CPU. The architectures have different memory systems, different ISAs, and different threading mechanisms, which make it very unlikely that performance tuning would arrive at the similar code configuration for these two architectures. For all the applications we have explored so far, this has always been the case. For example, the best performing matrix multiplication code uses per-block resources that are expressible in CUDA, but well over the hardware limits of current GPUs. The best CUDA code for the CPU uses 20KB of shared memory and 1024 logical threads per block, both over the hardware limits of current GPUs. Similarly, the best CP code uses an amount of constant memory larger than what a current GPU supports. Developing a system for tuning CUDA kernels to CPU architectures is a very interesting area of future work, both for programming practice and toolchain features.

For the benchmarks where the MCUDA performance is significantly below the best hand-tuned performance, we think that this is primarily because of algorithmic differences in the implementations. Projects like ATLAS have explored extensive code configuration searches far broader than we have considered in these experiments, and some of that work may be relevant here as well. People have achieved large speedups on GPU hardware with “unconventional” CUDA programming [15], and it is possible that more variations of CUDA code configurations may eventually bridge the current performance gap. The hand-tuned MRI-FHD implementation uses hand-vectorized SSE instructions across logical threads, whereas ICC vectorizes the innermost loop of the algorithm, seemingly with a minor reduction in efficiency. Future work should consider specifically targeting the thread loops for vectorization, and test the efficiency of such a transformation.

5 Related Work

With the initial release of the CUDA programming model, NVIDIA also released a toolset for GPU emulation [1]. However, the emulation framework was designed for debugging rather than for performance. In the emulation framework, each logical thread within a block is executed by a separate CPU thread. In contrast, MCUDA localizes all logical threads in a block to a single CPU thread for better performance. However, the MCUDA framework is less suitable for debugging the parallel CUDA application for two primary reasons. The first is that MCUDA modifies the source code before passing it to the compiler, so the debugger can not correlate the executable with the original CUDA source code. The second is that MCUDA enforces a specific scheduling of logical threads within a block, which would not reveal errors that could occur with other valid orderings of the execution of logical threads.

The issue of mapping small-granularity logical threads to CPU cores has been addressed in other contexts, such as parallel simulation frameworks [16]. There are also performance benefits to executing multiple logical threads within a single

CPU thread in that area. For example, in the Scalable Simulation Framework programming model, a CPU thread executes each of its assigned logical threads, jumping to the code specified by each in turn. Logical threads that specify suspension points must be instrumented to save local state and return execution to the point at which the logical thread was suspended. Taking advantage of CUDA's SPMD programming model and control-flow restrictions, MCUDA uses a less complex execution framework based on iteration within the originally threaded code itself. The technique used by MCUDA for executing logical threads can increase the compiler's ability to optimize and vectorize the code effectively. However, our technique is limited to SPMD programming models where each static barrier-wait instrinsic in the source code waits on a different thread barrier.

A large number of other frameworks and programming models have been proposed for data-parallel applications for multi-core architectures. Some examples include OpenMP [17], Thread Building Blocks [18], and HPF [19]. However, these models are intended to broaden a serial programming language to a parallel execution environment. MCUDA is distinct from these in that it is intended to broaden the applicability of a previously accelerator-specific programming model to a CPU architecture.

Liao et al. designed a compiler system for efficiently mapping the stream programming model to a multi-core architecture [20]. CUDA, while not strictly a stream programming model, shares many features with stream kernels. MCUDA's primary departure from mapping a stream programming model to multi-core architectures is the explicit use of data tiling and cooperative threading, which allows threads to synchronize and share data. With MCUDA, the programmer can exert more control over the kernels with application knowledge, rather than relying on the toolset to discover and apply them with kernel merging and tiling optimizations. It is also unclear whether the range of optimizations available in the CUDA programming model can be discovered and applied by an automated framework.

6 Conclusions

We have described techniques for efficiently implementing the CUDA programming model on a conventional multi-core CPU architecture. We have also implemented an automated framework that applies these techniques, and tested it on some kernels known to have high performance when executing on GPUs. We have found that for executing these translated kernels on the CPU, the expression of data locality and computational regularity in the CUDA programming model achieves much of the performance benefit of tuning code for the architecture by hand. These initial results suggest that the CUDA programming model could be a very effective way of specifying data-parallel computation in a programming model that is portable across a variety of parallel architectures.

As the mapping of the CUDA language to a CPU architecture matures, we expect that the performance disparity between optimized C code and optimized

CUDA code for the CPU will continue to close. As with any other level of software abstraction, there are more opportunities for optimization at lower levels of abstraction. However, if expressing computation in the CUDA language allows an application to be more portable across a variety of architectures, many programmers may find a slightly less than optimal performance on a specific architecture acceptable.

Acknowledgements

We would like to thank Micheal Garland, John Owens, Chris Rodrigues, Vinod Grover and NVIDIA corporation for their feedback and support. Sam Stone is supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. We acknowledge the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work was performed with equipment and software donations from Intel.

References

1. NVIDIA: NVIDIA CUDA, `http://www.nvidia.com/cuda`
2. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro 28(2) (in press, 2008)
3. Woop, S., Schmittler, J., Slusallek, P.: RPU: A programmable ray processing unit for realtime ray tracing. ACM Trans. Graph. 24(3), 434–444 (2005)
4. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual (May 2007)
5. Devices, A.M.: 3DNow! technology manual. Technical Report 21928, Advanced Micro Devices, Sunnyvale, CA (May 1998)
6. Ashcroft, E., Manna, Z.: Transforming 'goto' programs into 'while' programs. In: Proceedings of the International Federation of Information Processing Congress 1971, August 1971, pp. 250–255 (1971)
7. Kennedy, K., Allen, R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers, San Francisco (2002)
8. Lee, S., Johnson, T., Eigenmann, R.: Cetus - An extensible compiler infrastructure for source-to-source transformation. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, Springer, Heidelberg (2004)
9. Ayguadé, E., Blainey, B., Duran, A., Labarta, J., Martínez, F., Martorell, X., Silvera, R.: Is the schedule clause really necessary in OpenMP? In: Proceedings of the International Workshop on OpenMP Applications and Tools, June 2003, pp. 147–159 (2003)
10. Markatos, E.P., LeBlanc, T.J.: Using processor affinity in loop scheduling on shared-memory multiprocessors. In: Proceedings of the 1992 International Conference on Supercomputing, July 1992, pp. 104–113 (1992)
11. Hummel, S.F., Schonberg, E., Flynn, L.E.: Factoring: A practical and robust method for scheduling parallel loops. In: Proceedings of the 1001 International Conference of Supercomputing, June 1991, pp. 610–632 (1991)

12. Bull, J.M.: Feedback guidaed dynamic loop scheduling: Algorithms and experiments. In: European Conference on Parallel Processing, September 1998, pp. 377–382 (1998)
13. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D., Hwu, W.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (February 2008)
14. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S.Z., Stratton, J.A., Hwu, W.W.: Program optimization space pruning for a multithreaded GPU. In: Proceedings of the 2008 International Symposium on Code Generation and Optimization (April 2008)
15. Volkov, V., Demmel, J.W.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, CA (May 2008)
16. Cowie, J.H., Nicol, D.M., Ogielski, A.T.: Modeling the global internet. Computing in Science and Eng. 1(1), 42–50 (1999)
17. OpenMP Architecture Review Board: OpenMP application program interface (May 2005)
18. Intel: Threading Building Blocks, `http://threadingbuildingblocks.org/`
19. Forum, H.P.F.: High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University (May 1993)
20. Liao, S.W., Du, Z., Wu, G., Lueh, G.Y.: Data and computation transformations for Brook streaming applications on multiprocessors. In: Proceedings of the 4th International Symposium on Code Generation and Optimization, March 2006, pp. 196–207 (2006)