# MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores

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

{stratton, ssstone2, hwu}@crhc.uiuc.edu

IMPACT Technical Report

IMPACT-08-01

University of Illinois at Urbana-Champaign

Center for Reliable and High-Performance Computing

March 12, 2008

Revision: April 3, 2008

**Abstract**

The CUDA programming model, which is based on an extended ANSI C language and a runtime environment, allows the programmer to specify explicitly data parallel computation. NVIDIA developed CUDA to open the architecture of their graphics accelerators to more general applications, but did not provide an efficient mapping to execute the programming model on any other architecture.

This document describes Multicore-CUDA (MCUDA), a system that efficiently maps the CUDA programming model to a multicore CPU architecture. The major contribution of this work is the source-to-source translation process that converts CUDA code into standard C that interfaces to a runtime library for parallel execution. We apply the MCUDA framework to some CUDA applications previously shown to have high performance on a GPU, and demonstrate high efficiency executing these applications on a multicore CPU architecture. The thread-level parallelism, data locality and computational regularity of the code as expressed in the CUDA model achieve much of the benefit of hand-tuning an application for the CPU architecture. With the MCUDA framework, it is now possible to write data-parallel code in a single programming model for efficient execution on CPU or 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 for a GPU should not have to make an entirely separate programming effort to effectively parallelize the application across multiple CPU cores.

On the surface, most features included in the CUDA programming model seem relevant only to a specific GPU architecture. The programmer specifies data-parallel functions called *kernels*, expressed as parallel threads (hereafter referred to as *logical threads*) that execute cooperatively in thread blocks. At a kernel invocation, the programmer uses language extensions to specify runtime values for the number of thread blocks and number of logical threads per block. In the GPU architecture, these independent blocks are dynamically assigned to parallel processing units, where the logical threads are instantiated by hardware threading mechanisms and executed. The CUDA model also includes explicitly differentiated memory spaces to take advantage of specialized hardware memory resources. 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 blocks. The *shared* memory space maps to the scratchpad memory of the GPU, and is local to each thread block. The *texture* memory space uses the GPUs 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].

One might argue that the use of GPU specific features limits the execution of CUDA kernels to GPUs. Since the CPUs do not support these features, a CUDA kernel would not be executed efficiently on a CPU. This conclusion seems to be supported by the slow execution of CUDA kernels on CPUs using the emulation mode of the CUDA toolkit [1]. However, through the work reported in this paper, we show that CUDA kernels can be automatically translated into efficient code for multicore CPU execution. The use of the GPU-specific features in the CUDA model is actually beneficial to performance on the CPU, tbecause these features encourage the programmer to use more disciplined control flow and expose data locality.

In the CUDA model, logical threads within a block can have independent control flow through the program. However, for good performance on the GPU hardware, each thread should follow the same control flow, or execution trace, through the kernel code. The NVIDIA G80 GPU architecture executes logical threads in SIMD bundles called *warps*, but allows 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 group of logical threads with identical control flow. Therefore, CUDA programmers are strongly encouraged to adopt algorithms that force logical threads within a block to have very similar, if not exactly equivalent, execution traces. 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 constant and texture caches.

A closer viewing of the CUDA programming model suggests that there could also be an efficient mapping of the execution specified onto a commodity CPU multicore architecture. At the first granularity of parallelism, 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 it likely that the SIMD instructions common in current x86 processors [4] can be effectively used in many cases. 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 core's L1 data cache.

While the features of the model seem promising, the mapping of the computation is not straightforward. The conceptually easiest translation 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 previously noted, and incurs a large amount of scheduling overhead. Therefore, we propose a method of translating the CUDA program into an execution model that maintains the locality expressed in the programming model with existing operating system and hardware features.

There are several challenging goals in effectively translating these applications. First, each thread block should be scheduled to a single core for locality. 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 remainder of this document describes and analyzes the MCUDA system, which addresses these challenges and translates CUDA application into efficient parallel CPU programs. Section 2 describes the procedure for translating a CUDA kernel into an efficient block-level function. Section 3 describes the runtime framework that manages the execution of kernels. In Section 4 we discuss the performance of several kernels translated by the MCUDA framework. We discuss related work in Section 5, and make some concluding observations in Section 6.

```
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;
}
```

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

# 2 Kernel Translation

Automatic translation of the thread blocks is composed of a few key code transformations: 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 [5], but this is left for future work. All transformations are performed on the program's abstract syntax tree (AST).

## 2.1 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. This means that the implicit *threadIdx* variable now needs to be explicitly included, with control flow introduced to perform a logical thread's computation for each value of *threadIdx* within a single OS thread. An iterative structure around the entire code body, as shown in Figure 1, is a natural expression of the required additional control flow. For the remainder of the paper, we will consider this introduced iterative structure a *thread* *loop*. Each logical thread now corresponds to an iteration of this 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. Other implicit variables, such as *blockIdx*, are added to the parameter list of the function. Values for these variables are supplied by the runtime system when the function is called. If the kernel function contains no synchronization primitives, the translation of the control flow is complete. However, additional transformations are required to enforce programmer-specified synchronization points.

## 2.2 Enforcing synchronization with deep fission

For clarity in future discussion, we define a *synchronization statement* to be a statement or control structure in the program that all logical threads must enter and leave synchronously. This means that no logical thread can begin executing a synchronization statement before all other logical threads reach that synchronization statement, and all logical threads must complete the synchronization statement before any logical thread can continue past it. A thread loop is an instance of a synchronous statement, for example. A programmer-specified synchronization point is an example of a synchronization statement that contains no computation.

Because each logical thread is now a loop iteration, a loop fission transformation applied to the thread loop emulates the effects of a barrier synchronization across the logical threads. Loop fission applied to the thread loop at a certain point in the code forces each logical thread, in turn, to execute code up to that point, and then wait for all other logical threads to reach that point. This is exactly the behavior we expect from a barrier synchronization applied to the logical threads. Therefore, for synchronization points directly within the scope of the thread loop, we can enforce the synchronization by applying loop fission around that statement.

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. Barrier synchronization points within control structures cannot be enforced with loop fission, because a thread loop that begins outside that control structure cannot end withing the control structure without violating proper nesting. For example, in Figure 2(a), we cannot allow a thread loop to begin at the top of the function and end within the for loop.

```
__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

Figure 2: Applying deep fission in Matrix Multiplication to enforce synchronization.

To enforce these synchronization points, we take advantage of the CUDA programming model's requirement that control flow affecting synchronization points must be thread-independent within a block [6]. Since all threads in the thread block must synchronize within the control structure, all threads can be forced to enter and leave the control structure itself synchronously. This means that any control structure containing a synchronization point can be defined as a synchronization statement. In a transformation we call *deep fission*, we enforce synchronization statements within control structures by creating new thread loops within the scope containing those statements, and treating the scope itself as another synchronization statement, as shown in Figure 2(b-d). The full algorithm for enforcing synchronization statements is as follows.

1. For each synchronization statement, determine whether the immediate scope containing the synchronization statement is an instance of a thread loop.

2. If the condition is false, apply deep fission.

   (a) The scope containing the synchronization statement will itself become a new synchronization statement. If any expression within the control structure's declaration has side-effects in its evaluation, remove it. For instance, *for* loops such as the one in

```
thread_loop{
  …
  goto label;
  …

  __syncthreads();

  …
label:
  …
}
```

(a) Initial Code With Serialized Logical Threads

```
thread_loop{
  …
  goto label;
  …
}
//__syncthreads();
thread_loop{
  …
label:
  …
}
```

(b) Synchronize at Programmer Prompt

```
thread_loop{
  …
}
goto label;
thread_loop{
  …
}
//__syncthreads();
thread_loop{
  …
}
label:
thread_loop{
  …
}
```

(c) Synchronize at Control Flow Points

Figure 3: Addressing unstructured control flow. The goto statement and its target label are treated as additional synchronization statements for correctness.

Figure 2 must be translated into *while* loops with their initializing and update expressions included before and at the end of the loop body, respectively (see Figure 2(b)). Any conditions that have side effects must have their conditional evaluation expression moved out of the declaration and assigned to a temporary variable. The condition evaluation of the loop is then replaced by the temporary variable.

(b) Partition the scope containing the synchronization statement into two thread loops (Figure 2(c)).

(c) For an if-else construct, after partitioning the scope containing the synchronization, the scope defined by the other side of the condition must be wrapped in a thread loop construct as well to define the entire if-else construct as a synchronization statement.

(d) Define the scope of the current synchronization statement as a new synchronization statement. In the example of Figure 2(c), the while loop itself is marked as a new synchronization statement to force logical threads to enter and leave the loop synchronously, as expected by the enclosed thread loops. Return to step 1.

3. If the condition is true, apply a simple loop fission operation around the synchronization statement (see Figure 2(d)). If there are no more synchronization statements to be processed, the algorithm terminates. Otherwise, return to step 1 for the next synchronization statement.

After this algorithm has been applied with the list of programmer-specified synchronization points as input, the code may still have some control flow for which the algorithm has not properly accounted. For instance, control flow statements such as *continue*, *break*, or *goto* may not be handled correctly if the target of the control flow is within a different thread loop. Figure 3(b) shows a case where irregular control flow would result in incorrect execution. In some blocks, all logical threads may avoid the goto and synchronize correctly. In other blocks, all logical threads may take the goto, 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. Therefore, we define these early-exit and irregular control flow statements as synchronization statements as well. For such statements with one or more labels as their target, the target labels are also included in the synchronization statement list. The same algorithm for enforcing synchronization statements is then applied with this new list. For the example of Figure 3, this results in the code shown in Figure 3(c). This irregular control flow identification and synchronization 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 overspecify synchronization. This "oversynchronizing" 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 bound by synchronization statements 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.

## 2.3 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, software must emulate local storage for logical threads within the block. The simplest implementation creates an instance of the local variable with a separate memory location for each logical thread. 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. Functions with no synchronization 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 referenced outside the third thread loop.

Therefore, to use less memory space, the MCUDA framework only creates arrays for local variables that are referenced within more than one thread loop. This technique, called *selective replication*, results in the code shown in Figure 4(c), which allows all logical threads to use the same memory location for the local variable $k$. For future work, an even more selective approach could be defined with the use of a comprehensive live-variable analysis [5] to determine which variables never have a live value at the end of a thread loop.

References to a variable outside of the context of a thread loop can only exist in the definitions 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, offering similar performance benefit to the CPU cores.

```
__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

Figure 4: Data replication in Matrix Multiplication.

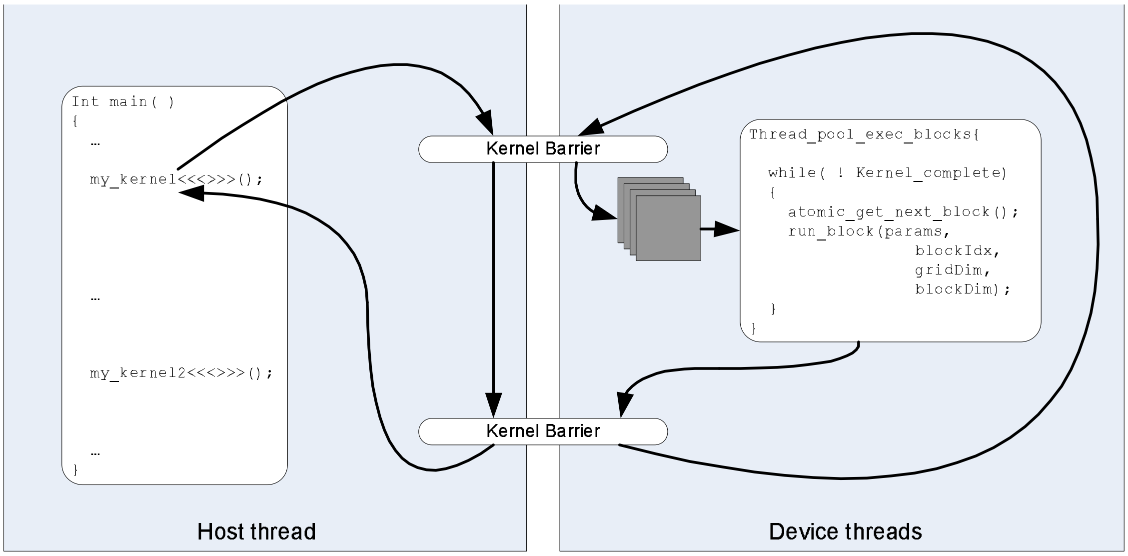


Figure 5: MCUDA runtime framework using dynamic block assignment

# 3 Work Distribution and Runtime Framework

To this point, the MCUDA framework has defined functions that, when invoked with a specific block index parameter, execute the full computation specified for that CUDA thread block. This section discusses how these thread blocks are executed in the current MCUDA runtime system to take advantage of multiple CPU cores.

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 indices 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, it is trivial to have multiple OS threads partition the set of block indices among themselves, and execute blocks concurrently on multithreaded CPU architectures. Many frameworks exist for such work distribution, such as OpenMP [7] or threading building blocks [8]. Our specific implementation uses POSIX threads as an example of how thread blocks can be efficiently scheduled. Figure 5 illustrates the runtime framework described in the remainder of this section.

In the host code, the kernel launch statement is translated into a function call to the runtime kernel launch routine. The function call specifies a reference to the kernel function to be invoked, the kernel configuration parameters, and the parameters to the kernel function itself. In the runtime library kernel launch routine, the host thread stores the kernel launch information into global variables, and enters a barrier synchronization point. A statically created pool of worker pthreads, representing the *device* in the CUDA model, also enters the barrier. On exiting, each worker thread reads the kernel launch data and begins executing blocks. The host thread then enters a second barrier to wait for kernel completion before returning to the host code.

The MCUDA runtime includes support for static and dynamic methods of assigning computation to CPU threads. The static method distributes a contiguous set of blocks to each worker thread. Any thread is assigned at most one additional block compared to any other thread. Each thread then executes independently until completing its set. Under the dynamic method, each worker thread iteratively acquires and executes blocks until all blocks in the kernel have been issued. Each OS thread, when requesting a block to execute, atomically loads the current block index, represented by a global variable. If it is within the range specified by the kernel launch configuration parameters, it executes that block, and increments the current block index to mark that the block is being processed. Otherwise, all blocks in the kernel have been issued.

In both methods, when each worker threads completes processing, it enters the barrier at which the host thread is waiting. When all worker threads reach the barrier, the kernel execution has completed, and the host thread is allowed to leave the barrier and return to the host code.

# 4 Performance Analysis

We have implemented the MCUDA automatic kernel translation framework under the Cetus source-to-source compilation framework [9], with slight modifications to the IR and preprocessor to accept ANSI C with the language extensions of CUDA. For compatibility with icc, library functions related to CUDA memory and runtime management were manually removed or replaced by standard libc functions. Figure 6 shows the kernel speedup of three applications: matrix multiplication of two 4kx4k element matrices, Coulombic Potential (CP), and MRI-FHD. These applications have previously shown to have very efficient CUDA implementations on a GPU architecture [10]. The CPU baselines that we are measuring against are the most heavily optimized CPU implementations

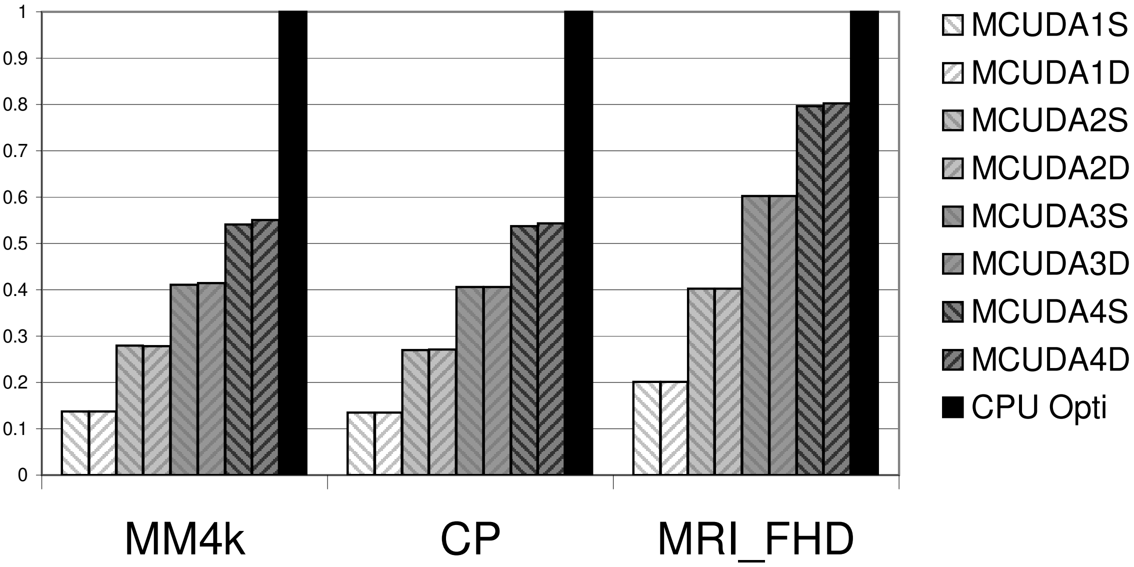


Figure 6: Performance (inverse runtime) of MCUDA kernels relative to optimized CPU code. MCUDA results vary by the number of worker threads (1-4) and the use of (S)tatic or (D)ynamic block scheduling. CPU Opti implementations are parallelized across 4 threads.

available to us, and are threaded by hand to make use of multiple CPU cores. All performance data was obtained on an Intel Core$^{TM}$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. Until the CPU is fully utilized, there are no noticeable performance differences between static and dynamic block scheduling policies. However, this is partially due to the structure of the applications themselves. MRI and CP both repeatedly invoke kernel functions, each of which is executed synchronously by the runtime system, limiting the amount of load imbalance that can accumulate within a single kernel invocation. Matrix multiplication uses a single kernel call for the entire computation, and shows more disparity between dynamic and static scheduling. In practice, we find that the dynamic method based on a work queue performs better than the static partitioning method, although only by a small margin for a small number of threads. We expect improved load balancing to increase the relative benefit of dynamic scheduling for a larger number of threads.

For each application, the performance of the CUDA code translated through the MCUDA framework is within a factor of two of the most optimized CPU implementation available. This suggests that the data tiling and locality expressed in effective CUDA kernels also gains most of the benefits of hand-optimization for the CPU architecture. The regularly structured iterative loops of the algorithm 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.

Typically, a CUDA kernel is tuned by manually applying variations to the kernel. These optimizations could include varying 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. Although we have not yet exhaustively explored the range of optimizations for these kernels, we have discovered some interesting results from experiments with a few different versions of the kernels for each application.

Our experiments show that not all optimizations that benefit a GPU architecture are effective for compiling and executing on the CPU. In the CP application, any degree of manual unrolling in the CUDA source code prevents the compiler from automatically applying vectorization optimizations using SSE/MMX instructions. Optimizations that spill local variables to shared memory were also ineffective, since the shared memory and local variables reside in the same memory space on the CPU.

We have also determined that the best optimization point for each application may be different depending on whether the kernel will execute on a GPU or CPU. Although we have not applied an exhaustive search to all of these applications yet, only matrix multiplication seems to have the same best configuration between the CPU and GPU implementations. For each of the others, we have verified that there is at least one configuration that outperforms the current GPU-optimal configuration when executed on the CPU with the current MCUDA framework. Determining how to optimize a CUDA kernel for the CPU architecture is a very interesting area of future work, both for programming practice and toolchain features.

For each of these benchmarks, we expect that the performance could potentially be tuned to more closely match the performance of tuned code in a C or assembly program. In CP, a large amount of redundant computation is done in the CUDA kernel to reveal additional fine-grained parallelism. However, manual tiling and loop invariant removal is the current method for reducing this redundant computation, which generates addressing patterns the compiler does not trust to use vectorization, and hence produces inferior performance at this time. In matrix multiplication, projects like ATLAS have explored extensive code configuration searches that can closely match MKL performance [11], and some of that work may be relevant here as well. The CPU code for MRI uses tuned loop unrolling and tiling factors that most likely account for the difference between MCUDA and hand-tuned C code.

# 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 more efficient 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 [12]. There are also performance benefit 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 original 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. The simplification of the control flow comes at the expense of completely independent control flow among the logical threads within a block.

A large number of other frameworks and programming models have been proposed for data-parallel applications for multicore architectures. Some examples include OpenMP [7], Thread Building Blocks [8], and HPF [13]. However, these frameworks 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 multicore architecture [14]. 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 multicore 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 automatically discovered and applied by an automated framework.

# 6 Conclusions

We have described techniques for efficiently implementing the CUDA programming model on a conventional multicore 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 decrease. 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.

## 7 Acknowledgments

We would like to thank Micheal Garland, John Owens 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 CUDA." http://developer.nvidia.com/object/cuda.html.

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, In press 2008.

[3] S. Woop, J. Schmittler, and P. Slusallek, "RPU: a programmable ray processing unit for realtime ray tracing," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 434–444, 2005.

[4] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2007.

[5] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Reading, MA: Addison-Wesley, 2006.

[6] NVIDIA Corporation, *CUDA Programming Guide*, February 2007.

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

[8] "Threading building blocks." http://threadingbuildingblocks.org/.

[9] S. Lee, T. Johnson, and R. Eigenmann, "Cetus - an extensible compiler infrastructure for source-to-source transformation," in *16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2003)*, 2003.

[10] 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 SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

[11] C. Whaley, P. A, and J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, vol. 27, pp. 3–25, September 2000.

[12] J. H. Cowie, D. M. Nicol, and A. T. Ogielski, "Modeling the global internet," *Computing in Science and Eng.*, vol. 1, no. 1, pp. 42–50, 1999.

[13] H. P. F. Forum, "High Performance Fortran language specification, version 1.0," Tech. Rep. CRPC-TR92225, Rice University, May 1993.

[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*, pp. 196–207, March 2006.