Compiler Support for High-level GPU Programming

Tianyi David Han and Tarek S. Abdelrahman
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
Email: {han,tsa}@eecg.toronto.edu

***Abstract*—We design a high-level abstraction of CUDA, called *hi*CUDA, using compiler directives. It simplifies the tasks in porting sequential applications to NVIDIA GPUs. This paper focuses on the design and implementation of a source-to-source compiler that translates a *hi*CUDA program into an equivalent CUDA program, and shows that the performance of CUDA code generated by this compiler is comparable to that of hand-written versions.**

I. Introduction

The Compute Unified Device Architecture (CUDA) has become a de facto standard for programming NVIDIA GPUs. However, CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host memory and various components of the GPU memory, and of manually optimizing the utilization of the GPU memory. To alleviate this burden, we design *hi*CUDA, a high-level directive-based language for CUDA programming. It abstracts away mechanical CUDA tasks into simple compiler directives, so that the programmer can perform these tasks easily and directly in the sequential code. *hi*CUDA supports the same programming paradigm already familiar to CUDA programmers. In this paper, we first give an overview of the *hi*CUDA language (detailed in [1]), but then focus on the design and implementation of a prototype *hi*CUDA compiler that translates a *hi*CUDA program into an equivalent CUDA program. Finally we discuss the evaluation of *hi*CUDA using this compiler and provide future research directions.

II. The *hi*CUDA Language

*hi*CUDA presents the programmer with a *computation* model and a *data* model. The computation model allows the programmer to identify code regions that are intended for execution on the GPU and to specify how they are to be executed in parallel. The data model allows programmers to allocate and de-allocate memory on the GPU and to move data back and forth between the host memory and the GPU memory. The remainder of this section gives a quick overview of the *hi*CUDA directives, which are described in details in [1].

The `kernel` directive identifies a code region for GPU execution. It specifies the name of the kernel to be created and the geometry of the thread grid that will execute the kernel, which can have arbitrary dimensionality. By default, the kernel region in its entirety is redundantly executed by every thread. The partitioning of kernel computation is specified using the `loop_partition` directive, which distributes loop iterations among GPU threads. It supports blocking and cyclic distributions over thread blocks, threads, or a combination of both. The `loop_partition` directive can be used to realize a wide range of partitioning schemes. It also supports non-perfect loop nests and loops with non-perfectly-divisible trip-count.

Since a GPU has its own unique memory hierarchy, GPU data must be explicitly managed. The `global`, `constant` and `shared` directives are for this purpose. Each directive manages the lifecycle of variables in the corresponding GPU memory. This lifecycle consists of allocation, data transfer and deallocation. Each operation is expressed in a single directive. The data transfer operation for a `global` or `constant` directive occurs between the host and the main GPU memory, and is executed by the host thread sequentially. However, data transfer for a `shared` directive occurs between the global and the shared memory on the GPU, and is done in parallel by many threads. All three directives maintain the property that no GPU memory variables are exposed to the programmer, which reduces programming burden and facilitates automatic management by the compiler. These directives also support the allocation and transfer of sections of arrays.

Figure 1 shows an example of applying *hi*CUDA directives to the standard matrix multiply code. For comparison purpose, the corresponding hand-written CUDA version is shown in Figure 2. It is clear that the *hi*CUDA code is simpler to write, to understand and to maintain. The programmer does not need to separate the kernel code from the host code nor to use explicit thread indices to partition computations.

```
float A[64][128], B[128][32], C[64][32];
// ... Randomly init A and B ...

#pragma hicuda global alloc A[*][*] copyin
#pragma hicuda global alloc B[*][*] copyin
#pragma hicuda global alloc C[*][*]

#pragma hicuda kernel matrixMul tblock(4,2) thread(16,16)
// C = A * B
#pragma hicuda loop_partition over_tblock over_thread
for (i = 0; i < 64; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 32; ++j) {
        float sum = 0;
        for (kk = 0; kk < 128; kk += 32) {
#pragma hicuda shared alloc A[i][kk:kk+31] copyin
#pragma hicuda shared alloc B[kk:kk+31][j] copyin
#pragma hicuda barrier
            for (k = 0; k < 32; ++k) {
                sum += A[i][kk+k] * B[kk+k][j];
            }
pragma hicuda barrier
#pragma hicuda shared remove A B
        }
        C[i][j] = sum;
    }
}
#pragma hicuda kernel_end

#pragma hicuda global copyout C[*][*]

#pragma hicuda global free A B C

printMatrix((float*)C, 64, 32);
```

Data allocation and initialization (host to GPU)

Kernel

Data write-back (GPU to host) and deallocation

Fig. 1. Matrix multiply example in *hi*CUDA.

III. The *hi*CUDA Compiler

We implemented a source-to-source *hi*CUDA compiler that takes as input a set of files containing C code with *hi*CUDA directives and produces an equivalent CUDA program in 3 files: one for the GPU

```
float A[64][128], B[128][32], C[64][32];
// ... Randomly init A and B ...
                                          Data allocation
                                          and initialization
                                          (host to GPU)
int size = 64 * 128 * sizeof(float);
cudaMalloc((void**)&d_A, size);
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
size = 128 * 32 * sizeof(float);
cudaMalloc((void**)&d_B, size);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
size = 64 * 32 * sizeof(float);
cudaMalloc((void**)&d_C, size);

dim3 dimBlock(16, 16);                    Kernel launch
dim3 dimGrid(32/dimBlock.x, 64/dimBlock.y);
matrixMul<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, 128, 32);

cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
cudaFree(d_A);
cudaFree(d_B);                            Data write-back (GPU to
cudaFree(d_C);                            host) and deallocation

/*********** matrixMul kernel ************/
__global__ void matrixMul(float *A, float *B, float *C,
        int wA, int wB)    // width of A and B
{
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int aBegin = wA * 16 * by + wA * ty + tx;
    int aEnd = aBegin + wA, aStep = 32;
    int bBegin = 16 * bx + wB * ty + tx, bStep = 32 * wB;

    __shared__ float As[16][32];
    __shared__ float Bs[32][16];

    float Csub = 0;
    for (int a = aBegin, b = bBegin; a < aEnd;
            a += aStep, b += bStep) {
        As[ty][tx] = A[a]; As[ty][tx+16] = A[a + 16];
        Bs[ty][tx] = B[b]; Bs[ty+16][tx] = B[b + 16*wB];
        __syncthreads();

        for (int k = 0; k < 32; ++k)
            Csub += As[ty][k] * Bs[k][tx];
        __syncthreads();
    }
    C[wB*16*by + 16*bx + wB*ty + tx] = Csub;
}
```

Fig. 2. Matrix multiply example in CUDA.

code (kernels), one for the CPU (host) code and a common header file. The generated CUDA code can be compiled to binary using the existing CUDA compiler tool chain from NVIDIA [2]. The *hi*CUDA compiler is built around Open64 (version 4.1) [3], and consists of three components: 1) a GNU 3 front-end, extended to support *hi*CUDA directive syntax, 2) a compiler phase that handles *hi*CUDA directives, and 3) a CUDA code generator, extended from the C code generator in Open64. The remainder of the section describes a number of key steps performed in the second component.

Kernel region identification. The compiler identifies regions of code that are surrounded by the `kernel` and `kernel_end` directives. For each kernel region, the compiler verifies that it has a single entry at the beginning of the region and a single fall-through exit. In addition, the compiler ensures that no kernel region is nested within another, emitting an error in such cases. Since kernel regions may appear in different functions in the code, an inter-procedural framework is necessary to do this error checking.

Management of GPU data. The compiler is responsible for generating code that manages the lifecycle of data residing on the device memory, as specified by the `global` and `constant` directives. For each `global` directive with an `alloc` clause, the compiler creates a new global memory variable and replaces the directive with an appropriate call to the CUDA run-time library that allocates global memory for this variable. The size of the allocated memory is based on the section specified in the `alloc` clause. Similarly, calls to the CUDA run-time library are inserted to de-allocate the global memory variables corresponding to the host variables specified by the `free` clause of a `global` directive. In the case of `constant` directives, the declaration of a constant memory variable is static, so its allocation and deallocation do not involve any code generation. The directives simply mark the scope of the constant memory variable.

For each `global` (or `constant`) directive with a `copyin` or `copyout` clause, the compiler inserts calls to the CUDA run-time library to transfer data between the host memory and the device memory. The clause specifies the data section of the host variable to be transferred. Since the CUDA run-time call can only transfer a *contiguous* chunk of data, the compiler must generate code that invokes the call multiple times (i.e., in a loop nest) for a non-contiguous data section, e.g. `A[*][1:2]` where `A` is a 5×5 matrix.

Reaching directives analysis. For each kernel region, the compiler must determine the GPU data, or the `global` and `constant` directives, that reach it. These directives may be defined in functions other than where the kernel itself is defined. Thus, an analysis similar to inter-procedural reaching definition analysis [4] is employed.

Data access analysis. The compiler must conservatively determine data accesses made by each kernel region. This includes scalars as well as sections of arrays. These data accesses are used to ensure that the data copied into the GPU memory by reaching `global` and `constant` directives cover the data needed by the kernel region (an error is emitted if otherwise). Note that the compiler can tolerate cases where a scalar read (not modified) by a kernel region is not explicitly copied into the GPU memory – this variable will be passed into the kernel as a parameter (which resides in the shared memory). Since the kernel region may contain function calls, the data access analysis must also be preformed inter-procedurally. The results of this analysis and the previous reaching directive analysis are combined to determine what GPU memory variables are needed by each kernel region. During this matching process, the compiler emits an error when ambiguity occurs, e.g. when a variable accessed in a kernel region is covered by both a `global` and a `constant` directive.

Outlining of kernel region. After data access analysis, the compiler extracts each kernel region into a separate kernel function. Its parameters are the GPU memory variables determined previously, which also include those scalar variables that are read-only in the kernel region. Then the compiler replaces the original code region by an invocation to the kernel function with an execution configuration specified by the `tblock` and `thread` clauses of the `kernel` directive. Since the thread block space and the thread space supported by CUDA is limited to be 2-D and 3-D respectively, the virtual spaces specified in the two clauses are first mapped to 2-D and 3-D. For example, `tblock(2,3,4,5)` is mapped to a 3-D thread block space `blockDim.(x,y,z) = (4x5,3,2)`.

Kernel access redirection. Once a kernel body is outlined into a separate function, accesses to scalars and arrays inside must be redirected to their corresponding GPU memory variables. Handling scalars is relatively straightforward as it only involves variable replacement. Handling an array access is a bit more involved since it is possible for only a section of the array to be allocated in the GPU memory. In such cases, the offset of each dimension of the access needs to be adjusted based on the allocated section's starting offset (for this dimension). Furthermore, access redirection must be done inter-procedurally since the scalars and arrays may be passed as arguments to functions called within the kernel body.

Kernel loop partitioning. The compiler translates each `loop_partition` directive in a kernel function, by modifying the bounds and step of the associated loop so that they represent the iterations executed by each GPU thread. For example, distributing a simple loop `for (i = 0; i < 5; ++i)` among 3 threads (cyclically) results in loop `for (i = threadIdx.x; i <`

`5; i += 3)` in the kernel function. The code generation is straightforward in most cases, except when the loop iterations are distributed among thread blocks in a `BLOCK` fashion. In such a case, if the compiler is not certain that the distribution is even, it must insert code to prevent certain thread blocks from executing extra iterations. As an optimization, if the compiler is certain that each thread executes at most one iteration of the loop, it eliminates the loop, and optionally replaces it with a conditional branch.

The number of thread blocks and threads to which a loop is distributed is determined by the geometry of the thread block and thread spaces of the enclosing kernel region, and the nesting level of the `loop_partition` directive. Since `loop_partition` directives may be placed in functions called within a kernel region, an interprocedural propagation of kernel contextual information is required.

Utilizing the shared memory. The translation of `shared` directives is more complicated than that of `global` and `constant` directives. First, when declaring a shared memory variable, the compiler cannot simply determine the variable shape based on the array section specified in the directive. It must merge this section across all iterations of the enclosing loops that are concurrently executed by a thread block. This ensures that the shared memory variable can host data needed by all concurrent threads. Since the `shared` directives and the enclosing loops may not be in the same function, an interprocedural propagation of loop contextual information is required to do section merging. Second, data transfer between the shared memory and the global memory is done cooperatively by all threads in a thread block. The compiler determines an assignment of array elements accessed by each thread, and optionally pads the shared memory region so that bank conflict is avoided.

Pointer promotion. To support the use of dynamically allocated arrays, *hi*CUDA provides a `shape` directive that allows the programmer to specify the shape of such arrays. The compiler first propagates this information inter-procedurally and then promotes all array references made through pointers (e.g., `*p`) into regular array references (i.e., `p[0]`). This allows the above data analyses to be used with minimal modification. The process of pointer promotion involves factoring 1-D access offsets into multi-dimensional offsets, which must be verified to be within the corresponding array bounds using a linear programming solver.

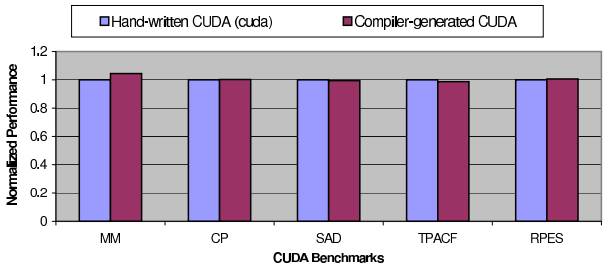

Fig. 3. Performance comparison of *hi*CUDA code and CUDA code.

It is clear that many of the above steps must be done inter-procedurally. Thus, we have extended the inter-procedural analysis framework of the Open64 infrastructure to provide this functionality. Our analysis first classifies each function in the input code into one of four classes: *K-function*, *MK-function*, *IK-function* and *N-function*, by doing a simple inter-procedural propagation. K-functions are those that directly contain kernel regions; MK-functions are those that do not contain kernel regions but may reach one through a chain of function calls; IK-functions are those that may be reached through a chain of function calls starting within a kernel region; and N-functions are the rest. With this classification scheme in place, kernel-related error checking becomes straightforward, and the compiler can perform various inter-procedural analyses on specific classes of functions. For example, reaching directives analysis is done among K-functions and MK-functions; access redirection is done among K-functions and IK-functions.

IV. Experimental Evaluation

Using the prototype compiler, we evaluated *hi*CUDA in two aspects: 1) *performance*, to ensure that using *hi*CUDA results in comparable performance as writing CUDA code manually, and 2) *usability*. To evaluate the performance of *hi*CUDA, we used five CUDA benchmarks (matrix multiply and four benchmarks from the Parboil suite [5]). Starting from the sequential version of each benchmark, we first applied high-level optimizations performed for the corresponding CUDA version, such as loop unrolling, loop collapsing, data repacking and use of device-specific math functions. We then inserted *hi*CUDA directives to achieve the same parallelization scheme as the CUDA version. As shown in Figure 3, for all benchmarks, the performance (i.e. the inverse of the wall-clock execution time) of the compiler-generated CUDA code is within 2% of that of the hand-written CUDA versions. To assess the usability of *hi*CUDA, we provided the prototype compiler to a medical research group at University of Toronto to accelerate a real-world application: Monte Carlo simulation for multi-media tissue [6]. A manually written CUDA version was developed in 3 months, achieving a 27X speedup on a Geforce 8800GTX card over an Intel Xeon (3.0GHz). The same code transformations and optimizations were then applied to the original version using *hi*CUDA directives, achieving the same speedup within 4 weeks.

V. Conclusions

We have designed a high-level abstraction of CUDA, called *hi*CUDA, that uses simple compiler directives. Using a prototype source-to-source compiler, we have shown that *hi*CUDA does not sacrifice performance for ease-of-use. Further, the initial use of *hi*CUDA for a real-world application suggests that it can significantly cut development time.

The current design of *hi*CUDA aims to simplify the most common tasks in CUDA programming. It is a starting point for ongoing research in high-level GPU programming. We have observed that the GPU implementation of many applications involve standard loop transformations and high-level idioms, such as reduction, histogram and scan. Since they involve non-trivial code changes, it is beneficial to incorporate them into *hi*CUDA. Further, the capability of the *hi*CUDA compiler can be enhanced to guide programmers to correct and optimize programs. Finally, the use of some *hi*CUDA directives can be automated to reduce the burden on programmers. This direction would lead to a parallelizing compiler for GPU, which requires no intervention on the part of the programmer.

References

[1] T. D. Han and T. S. Abdelrahman, "hiCUDA: a high-level directive-based language for GPU programming," in *GPGPU-2*, 2009, pp. 52–61.

[2] NVIDIA, "The CUDA Compiler Driver NVCC v1.1," http://www.nvidia.com/object/cuda_programming_tools.html, 2007.

[3] "Open64 research compiler," http://open64.sourceforge.net.

[4] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[5] I. R. Group, "The Parboil benchmark suite," http://www.crhc.uiuc.edu/IMPACT/parboil.php, 2007.

[6] L. Wang, S. L. Jacques, and L. Zheng, "MCML–Monte Carlo modeling of light transport in multi-layered tissues," *Computer Methods and Programs in Biomedicine*, vol. 47, no. 2, pp. 131–146, 1995.