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, ts}@eecg.toronto.edu

Abstract—We design a high-level abstraction of CUDA, called hiCUDA, 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 hiCUDA 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 hiCUDA, 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. hiCUDA supports the same programming paradigm already familiar to CUDA programmers. In this paper, we first give an overview of the hiCUDA language (detailed in [1]), but then focus on the design and implementation of a prototype hiCUDA compiler that translates a hiCUDA program into an equivalent CUDA program. Finally we discuss the evaluation of hiCUDA using this compiler and provide future research directions.

II. THE HICUDA LANGUAGE

hiCUDA 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 hiCUDA 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 hiCUDA 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 hiCUDA 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.

```c
# pragma hiCUDA kernel matrixmul1 bblock(4,2) thread(16,16) { 
# pragma hiCUDA global alloc A[64][128], B[128][32], C[64][32]; 
// ... Randomly init A and B ... 
float A[64][128], B[128][32], C[64][32]; 

// Internal variables 

# pragma hiCUDA global alloc A[*][*] copyin 
# pragma hiCUDA global alloc B[64][*] copyin 
# pragma hiCUDA global alloc C[*][*] copyin 

data allocation and initialization (host to GPU)

# pragma hiCUDA kernel matrixmul1 bblock(4,2) thread(16,16) { 
# pragma hiCUDA loop_partition over_bblock over_thread 
for (i = 0; i < 64; i++) { 
# pragma hiCUDA loop_partition over_bblock over_thread 
for (j = 0; j < 32; j++) { 
    float sum = 0; 
    for (k = 0; k < 128; k++) sum += A[i][kk+k] * B[kk+k][j]; 
    C[i][j] = sum; 
} 
} 

// Data write-back (GPU to host) and deallocation

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

Fig. 1. Matrix multiply example in hiCUDA.

III. THE HICUDA COMPILER

We implemented a source-to-source hiCUDA compiler that takes as input a set of files containing C code with hiCUDA directives and produces an equivalent CUDA program in 3 files: one for the GPU...
Fig. 2. Matrix multiply example in CUDA.

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

int size = 64 * 128 + sizeof(float);
cudaMallocHost(void** A, size);
cudaMemsetHost((void*)A, 0, size);
size = 128 * 32 + sizeof(float);
cudaMallocHost(void** B, size);
cudaMemsetHost((void*)B, 0, size);
size = 64 * 32 + sizeof(float);
cudaMallocHost(void** C, size);

dim3 dimBlock(16, 16);
dim3 dimGrid(5, 5, 5)/KernGird/;
matrixmul<<dimGrid, dimBlock>>(A, B, C, 64, 128, 32);
cudaFree(A);
cudaFree(B);
cudaFree(C);

__global__ void matrixmul(float *A, float *B, float *C, int wa, int wb) {
    int bx = blockIdx.x * blockDim.x + threadIdx.x;
    int tx = threadIdx.y + blockDim.y * blockIdx.y;
    int bain = wa + (tx + wa) * tx;
    int ain = bain + wa, bstep = 32;
    __shared__ float A_shared[64][32];
    __shared__ float B_shared[32][64];
    float Cshared = 0;
    for (int a = abegin; a < ainend; a += astep, b += bstep)
        for (int k = 0; k < 32; k++)
            Cshared += A_shared[a][k] * B_shared[k][tx];
    __syncthreads();
}
```

Fig. 2. Matrix multiply example in CUDA.
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 \textit{hiCUDA} in two aspects: 1) \textit{performance}, to ensure that using \textit{hiCUDA} results in comparable performance as writing CUDA code manually, and 2) \textit{usability}. To evaluate the performance of \textit{hiCUDA}, we used five CUDA benchmarks (matrix multiply and four benchmarks from the Parboil suite \cite{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 \textit{hiCUDA} 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 \textit{hiCUDA}, 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 \cite{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 \textit{hiCUDA} directives, achieving the same speedup within 4 weeks.

V. CONCLUSIONS

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

The current design of \textit{hiCUDA} 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 \textit{hiCUDA}. Further, the capability of the \textit{hiCUDA} compiler can be enhanced to guide programmers to correct and optimize programs. Finally, the use of some \textit{hiCUDA} 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