Bridging the GPGPU Programmability Gap

Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
University of Toronto

The *hi*CUDA Project

Ease the programming of Graphics Processing Units (GPUs) using compiler directives (pragams)

Tianyi David Han

Why Not CUDA

- Many “mechanical” steps:
  - Packaging of kernel functions
  - Using thread index variables to partition computation
  - Managing data in GPU memories
- Can become tedious and error-prone
  - Particularly when repeated many times for optimizations
- Difficult for programmers to understand, debug and code maintain; for compilers to optimize
  - Hard to “see the big picture” of the computation

*hi*CUDA*: hi*-level CUDA

- A directive-based language that maintains the CUDA programming model
  - #pragma hicuda <directive name> [<clauses>]+

  Programmers can perform common CUDA tasks directly to the sequential code, with a few lines of directives

- Keeps the structure of the original code
  - Easier to understand and maintain
  - Eases experimentation

*hi*CUDA by Example – MM

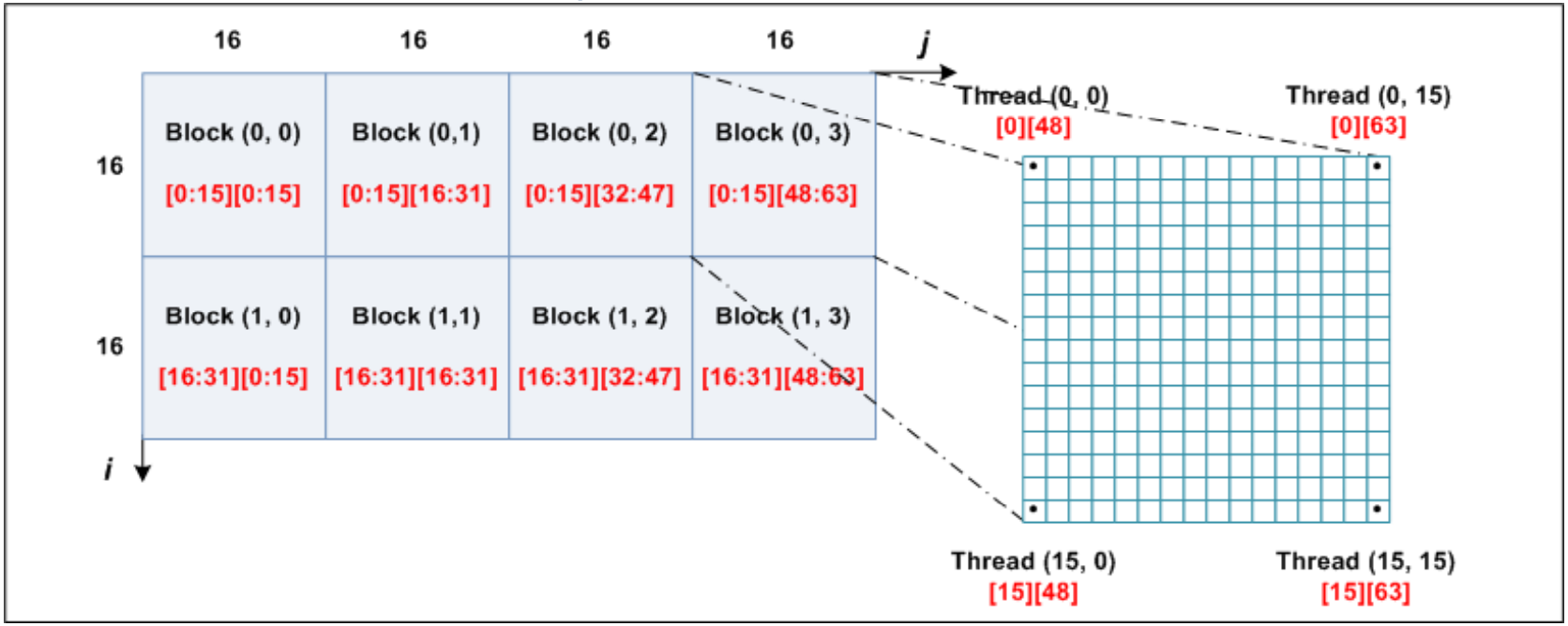
```
float A[32][96], B[96][64], C[32][64];
for (i = 0; i < 32; ++i) {
    for (j = 0; j < 64; ++j) {
        float sum = 0;
        for (k = 0; k < 96; ++k) sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
```

Kernel Identification

```
float A[32][96], B[96][64], C[32][64];
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
for (i = 0; i < 32; ++i) {
    for (j = 0; j < 64; ++j) {
        float sum = 0;
        for (k = 0; k < 96; ++k) sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
#pragma hicuda kernel_end
```

Computation Partitioning

```
float A[32][96], B[96][64], C[32][64];
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
#pragma hicuda loop_partition over_tblock over_thread
for (i = 0; i < 32; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 64; ++j) {
        float sum = 0;
```



GPU Data Management

```
float A[32][96], B[96][64], C[32][64];
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
#pragma hicuda loop_partition over_tblock over_thread
for (i = 0; i < 32; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 64; ++j) {
        float sum = 0;
        for (k = 0; k < 96; ++k) sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
#pragma hicuda kernel_end
```

GPU Data Management

```
float A[32][96], B[96][64], C[32][64];
#pragma hicuda global alloc A[*][*] copyin
#pragma hicuda global alloc B[*][*] copyin
#pragma hicuda global alloc C[*][*]
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
#pragma hicuda loop_partition over_tblock over_thread
for (i = 0; i < 32; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 64; ++j) {
        float sum = 0;
        for (k = 0; k < 96; ++k) sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
#pragma hicuda kernel_end
#pragma hicuda global copyout C[*][*]
#pragma hicuda global free A B C
```

Utilizing Shared Memory

```
float A[32][96], B[96][64], C[32][64];
...
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
#pragma hicuda loop_partition over_tblock over_thread
for (i = 0; i < 32; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 64; ++j) {
        float sum = 0;
        for (k = 0; k < 96; ++k) sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
#pragma hicuda kernel_end
...
```

Utilizing Shared Memory

```
float A[32][96], B[96][64], C[32][64];
...
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
#pragma hicuda loop_partition over_tblock over_thread
for (i = 0; i < 32; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 64; ++j) {
        float sum = 0;
#pragma hicuda shared alloc A[i][*] copyin
#pragma hicuda shared alloc B[*][j] copyin
#pragma hicuda barrier
        for (k = 0; k < 96; ++k) sum += A[i][k] * B[k][j];
#pragma hicuda barrier
#pragma hicuda shared remove A B
        C[i][j] = sum;
    }
}
#pragma hicuda kernel_end
...
```

*hi*CUDA – MM

```
float A[32][96], B[96][64], C[32][64];
#pragma hicuda global alloc A[*][*] copyin
#pragma hicuda global alloc B[*][*] copyin
#pragma hicuda global alloc C[*][*]
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
#pragma hicuda loop_partition over_tblock over_thread
for (i = 0; i < 32; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 64; ++j) {
        float sum = 0;
#pragma hicuda shared alloc A[i][*] copyin
#pragma hicuda shared alloc B[*][j] copyin
#pragma hicuda barrier
        for (k = 0; k < 96; ++k) sum += A[i][k] * B[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
```

CUDA – MM Kernel

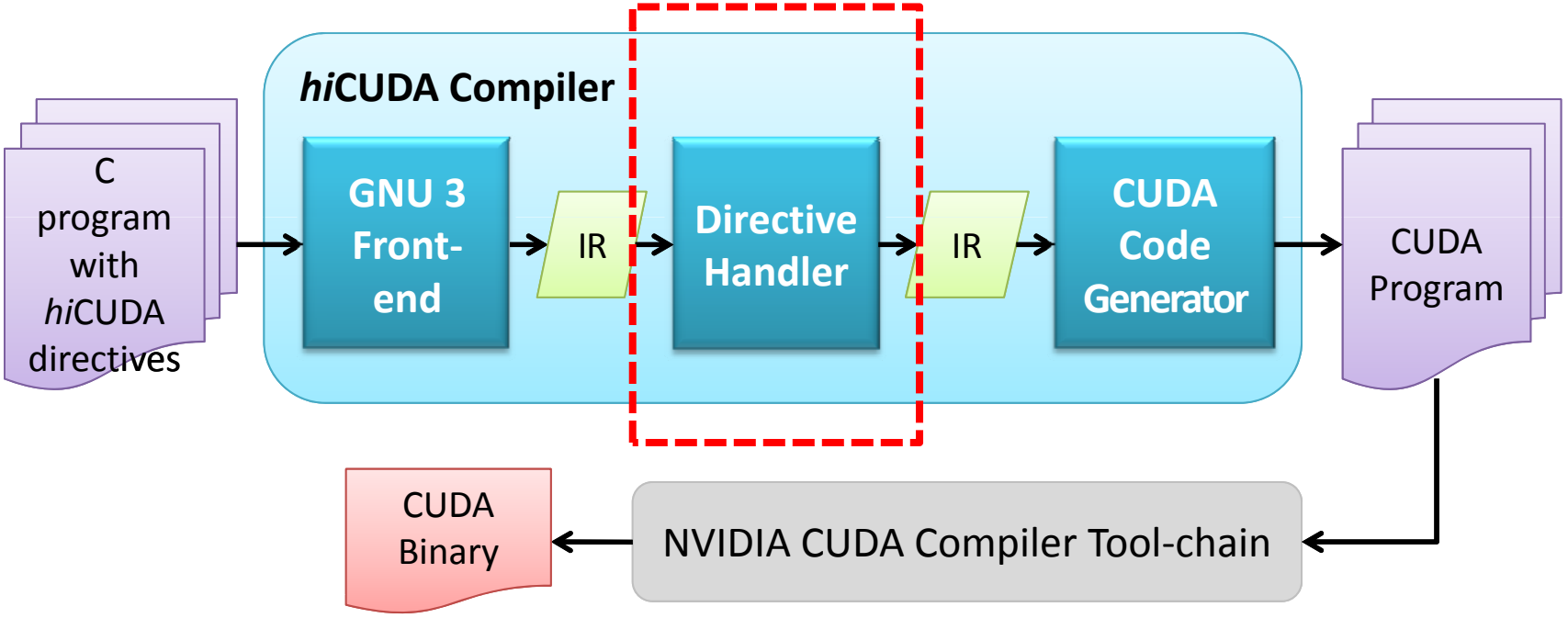
```
__global__ void matrixMul(float *A, float *B, float *C)
{
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;

    int aBegin = 96 * (16 * by + ty) + tx;
    int bBegin = 64 * ty + (16 * bx + tx);

    __shared__ float As[16][96];
    __shared__ float Bs[96][16];

    float Csub = 0;

    for (int i = 0; i < 6; ++i) {
        As[ty][tx + 16 * i] = A[aBegin + 16 * i];
        Bs[ty + 16 * i][tx] = B[bBegin + 16 * wB * i];
    }
    __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;
}
```

*hi*CUDA Directives

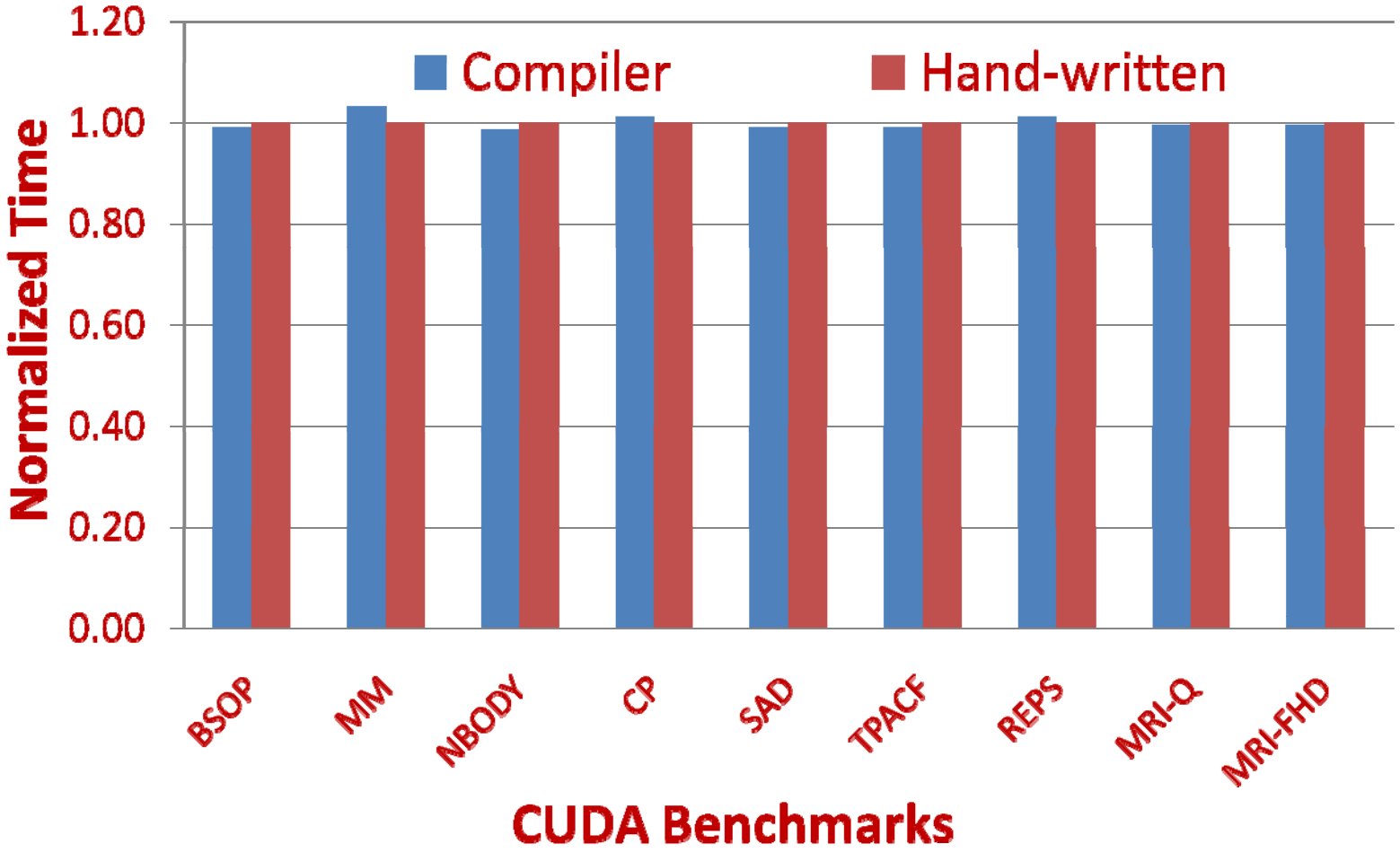
- Support modular programming
- Support dynamically allocated arrays
- Support array sections
- Other directives

The *hi*CUDA Compiler

- Source-to-source translator
- Built around open64



Performance Evaluation



MCML Case Study

- Use *hi*CUDA to accelerate Monte Carlo simulation for Multi-Layered media (MCML)
  - ~800 lines of kernel code (if fully-inlined)
- Initial version: 17 *hi*CUDA directives
  - same speedup (18X) as the hand-written CUDA version
  - 3 weeks vs. 3 months
- Lesson learned
  - Support the use of function calls in kernel regions!

Related Work

- OpenMP to GPGPU (R. Eigenmann/Purdue)
  - Leverages a standard
  - Not control over shared memory
- Accelerator Programming Model (PGI)
  - Similar directives
  - No control over shared memory
  - No support for modularity
- HMPP (PathScale/CAPS Enterprise)
  - Modularity?

Future Work

- Directives for common operations
  - Reductions, inductions, scans and histograms, stencils, etc.
  - GPU math operations
  - Some loop transformations (e.g., strip mining)
- User studies
- Support for openCL

*hi*CUDA Release

- The *hi*CUDA compiler is released as open source at:

  www.hicuda.org

- Documentation and a tutorial

- Web access to compiler/GPU

- Use and feedback is welcomed