*hi*CUDA: A High-level Directive-based Language for GPU Programming

David Han
Supervisor: Tarek Abdelrahman

University of Toronto
May 20, 2009

Outline

- Motivation of *hi*CUDA
- *hi*CUDA through an example
- Experimental evaluation
- Conclusions
- Future work

Motivation

- CUDA: a C-extended language for programming NVIDIA Graphics Processing Units
- 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
- Make programs difficult to understand, debug and maintain

High-level CUDA (*hi*CUDA)

- A directive-based language that maintains the CUDA programming model

```
#pragma hicuda <directive name> [<clauses>]+
```

- Programmers can perform common CUDA tasks directly into the sequential code, with a few directives
  - Keeps the structure of the original code, making it more comprehensible and easier to maintain
  - Eases experimentation with different code configurations

CUDA vs. *hi*CUDA

Typical CUDA programming steps

1. Identify and package a kernel
2. Partition kernel computation among a grid of GPU threads
3. Manage data transfer between the host memory and the GPU memory
4. Perform memory optimizations

*hi*CUDA directives

1. **kernel**
2. **loop_partition**
3. **global, constant**
4. **shared**

An Example: Matrix Multiply

```
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;
    }
}
```

Standard matrix multiplication algorithm

Kernel identification

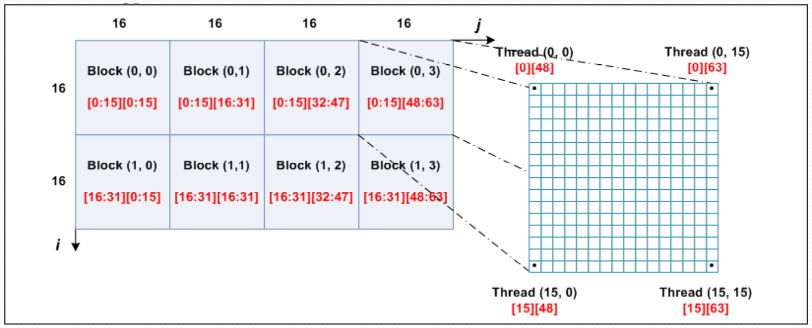
```
float A[32][96], B[96][64], C[32][64];
for (i = 0; i < 32; ++i) {
    for (j = 0; j < 64; ++j) {
        float sum = 0;
        for (k = 0; k < 96; ++k) sum += A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
```


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) {
```



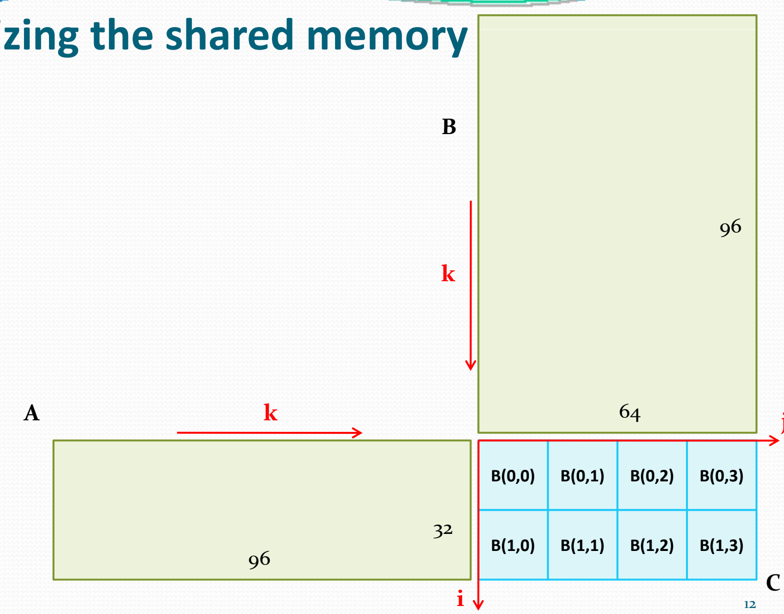
GPU data management

```
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
```

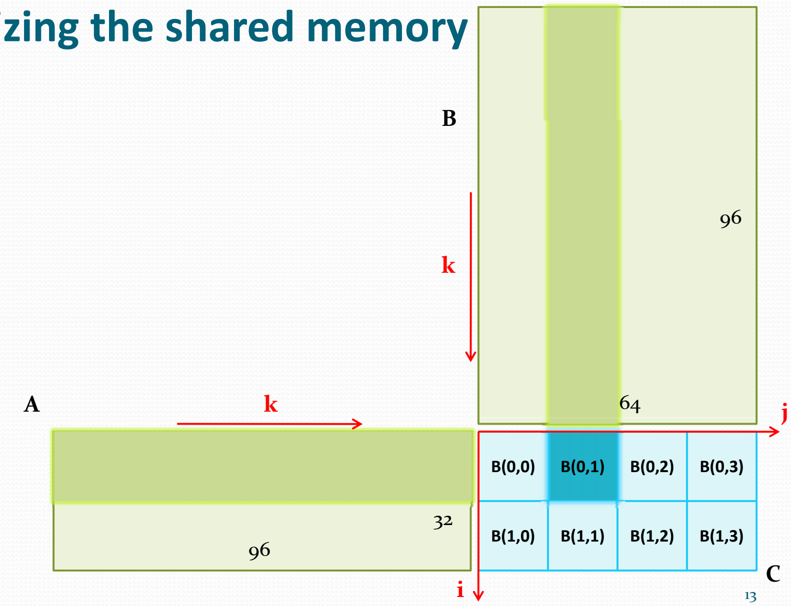

GPU data management

```
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
```

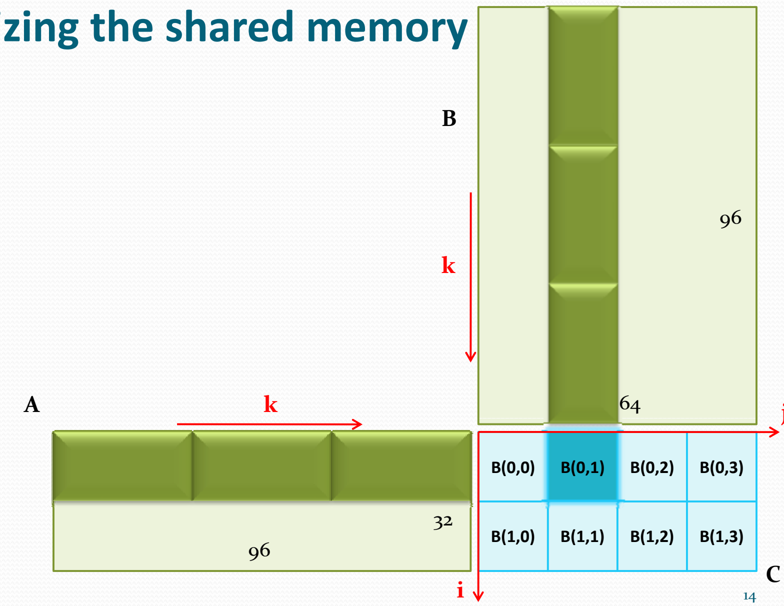

Utilizing the shared memory



Utilizing the shared memory



Utilizing the shared memory



Utilizing the shared memory

```
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 the shared memory

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

Strip-mine loop *k*

Utilizing the shared memory

```
float sum = 0;
for (kk = 0; kk < 96; 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;
```

Add the **shared** directives

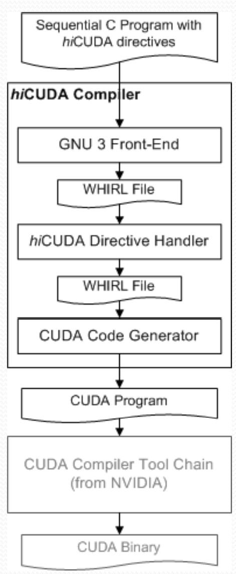
Evaluation of *hi*CUDA

- We have developed a prototype *hi*CUDA compiler for translation into CUDA programs
- We evaluated the performance of *hi*CUDA programs against manually written CUDA programs
  - Four benchmarks from the *Parboil* suite (UIUC Impact Research Group)
- User assessment on *hi*CUDA
  - Monte Carlo simulation for Multi-Layer media (MCML)

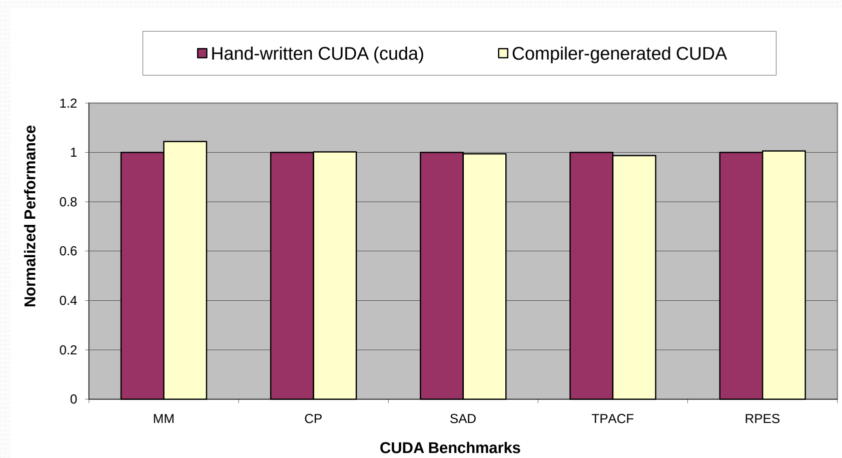
*hi*CUDA Compiler

- Source-to-source
- Based on Open64 (v4.1)

- Kernel outlining
  - Array section analysis (inter-procedural)
  - Data flow analysis
- Distribution of kernel loops
  - Data dependence analysis
- Access redirection inside kernels
  - Array section analysis
- Generation of optimized data transfer code
  - Auto-pad shared memory variables for bank-conflict-free transfers



Performance Evaluation



Ease of Use

- Used by a medical research group at University of Toronto, in accelerating Monte Carlo simulation for Multi-Layer media (MCML)
- CUDA version was developed in 3 months, while *hi*CUDA version was developed in 4 weeks
  - Both include the learning phase
- Disclaimer

Conclusions

- *hi*CUDA provides a high-level abstraction of CUDA, through compiler directives
  - No explicit creation of kernel functions
  - No use of thread index variables
  - Simplified management of GPU data
- We believe *hi*CUDA results in:
  - More comprehensible and maintainable code
  - Easier experimentation with multiple code configurations
- Promising evaluation using our prototype compiler

Future Work

- Finalize and release the *hi*CUDA compiler, to be available at:

  www.hicuda.org

- Assess and evolve the language design based on feedback
  - High-level programming patterns/idioms, such as reduction, histogram, etc.
- Explore compiler analyses and optimizations for automatic generation of *hi*CUDA directives