**CUDA is Nice**

- CUDA: a C-extended language for programming NVIDIA Graphics Processing Units:
  - Close to the hardware: exposes processors and the memory hierarchy
  - Simple extensions to C: `__global__`, `__device__`, `<<< ... >>>`, etc.
  - Runtime libraries
  - Etc.

**But …**

- 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

**Outline**

- What is hiCUDA and why
- hiCUDA through an example
- hiCUDA directives
- Prototype and experimental evaluation
- Concluding remarks
High-level CUDA (hiCUDA)

- A directive-based language that maintains the CUDA programming model
  ```cpp
  #pragma hicuda <directive name> [ clauses ]
  ```
- Programmers can perform common CUDA tasks directly into the sequential code, with a few directives
  ```cpp
  #pragma hicuda <directive name> [ clauses ]
  ```
- Keeps the structure of the original code, making it more comprehensible and easier to maintain
- Eases experimentation with different code configurations

CUDA vs. hiCUDA

**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

**hiCUDA directives**

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

An Example: Matrix Multiply

```cpp
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

```cpp
#global__ void matrixMul(float *A, float *B, float *C, int wA, int wB)
{
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int aBegin = wA * 16 * by + wA * ty + tx, 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*tx + wB*ty + tx] = Csub;
}
```

Matrix Multiply Kernel in CUDA
main function in CUDA

```c
int main(int argc, char **argv)
{
    float *d_A, *d_B, *d_C;
    int size = 32 * 96 * sizeof(float);
    cudaMalloc((void**)&d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    size = 96 * 64 * sizeof(float);
    cudaMalloc((void**)&d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    size = 32 * 64 * sizeof(float);
    cudaMalloc((void**)&d_C);
    dim3 dimBlock(16, 16);
    dim3 dimGrid(2, 4);
    matrixMul<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, 96, 64);
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Kernel identification

```c
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

```c
float A[32][96], B[96][64], C[32][64];
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
#pragma hicuda kernel_end
```

© David Han and Tarek Abdelrahman 4/4/2009
Computation partitioning

```c
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;
```

GPU data management

```c
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;
```

Utilizing the shared memory

```c
#pragma hicuda global copyout C[*][*]
#pragma hicuda global free A B C
```
Utilizing the shared memory

float A[32][96], B[96][64], C[32][64];
#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) {
    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) {
    #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

Matrix Multiply Kernel in hiCUDA

#ifdef __CUDA_ARCH__

__global__ void matrixMul(float *A, float *B, float *C, int wA, int wB) {
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    __shared__ float As[16][32]; __shared__ float Bs[32][16];
    float Csub = 0;
    for (int a = aBegin, b = bBegin; a < aEnd; a += aStep) {
        __shared__ float As[16][32]; __shared__ float Bs[32][16];
        float Csub = 0;
        for (int b = bBegin, a = aBegin; 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;
    }
}
#endif
**Summary of hiCUDA Directives**

**Computation Model**
- `kernel` directive
- `loop_partition` directive
- `barrier` directive
- `singular` directive

**Data Model**
- `global` directive
- `shared` directive
- `constant` directive
- `shape` directive

**kernel directive**
- Allow arbitrary dimensionality of the thread space
  - `kernel matrixMul tblock(2,4,6) thread(16,4,4)`
- Support asynchronous kernel execution
  - `kernel matrixMul tblock(2,4) thread(16,16) nowait`
- The *entire* kernel region is executed by *every* thread

**loop_partition directive**
- Can be arbitrarily nested
  - The nesting level determines the associated dimension of the tblock/thread space
- `over_tblock` and `over_thread` clauses are optional
  - `loop_partition over_tblock`
    - Each thread executes *all* iterations assigned to the thread block
  - `loop_partition over_thread`
    - Each thread block executes *all* iterations
- Support **BLOCK/CYCLIC** distribution of loop iterations

```c
#pragma hicuda kernel test tblock(2) thread(4)
#pragma hicuda loop_partition over_tblock(CYCLIC) over_thread
for (int i = 0; i < 16; ++i) { ... }
#pragma hicuda kernel_end
```
**Loop Partition Directive**

- Support non-uniform distribution of iterations
  - Automatically generate “guard code”

```c
#pragma hicuda kernel test tblock(2) thread(4)
#pragma hicuda loop_partition over_tblock(CYCLIC) over_thread
for (int i = 0; i < 13; ++i) { ... }
#pragma hicuda kernel_end
```

**Support non-perfect loop nests**

```c
#pragma hicuda loop_partition over_tblock(8)
for (i = 0; i < 32; ++i) {
  // ... Code A ...
  #pragma hicuda loop_partition over_thread(2)
  for (j = 0; j < 64; ++j) {
    // ... Code B ...
  }
}
```

**Singular Directive**

- Redundant execution may not be desirable

```c
#pragma hicuda loop_partition over_tblock(8)
for (i = 0; i < 32; ++i) {
  Arr[0] = 0;
  #pragma hicuda loop_partition over_thread(2)
  for (j = 0; j < 64; ++j) {
    // ... Code B ...
  }
}
```
**hiCUDA data directives**

- No GPU memory variable is exposed to the programmer

- Support the use of dynamic arrays
  
  - `shape` directive
    
    ```c
    #define N_ELEMS 100
    int* ptr = (int*)malloc(N_ELEMS*sizeof(int));
    #pragma hicuda shape ptr[N_ELEMS]
    #pragma hicuda global alloc ptr[*] copyin ptr[1:98]
    ```

**hiCUDA data directives**

- Support allocation and transfer of array sections
- Example: *jacobi*

```c
float A[66][66], B[66][66];
for (iter = 0; iter < 10; ++iter)
{
    for (i = 1; i <= 64; ++i)
        for (j = 1; j <= 64; ++j)
    for (i = 1; i <= 64; ++i)
        for (j = 1; j <= 64; ++j)
            A[i][j] = B[i][j];
}
```

**hiCUDA data directives**

- Support allocation and transfer of array sections
- Example: *jacobi*

```
float A[66][66], B[66][66];
for (iter = 0; iter < 10; ++iter)
{
    for (i = 1; i <= 64; ++i)
        for (j = 1; j <= 64; ++j)
    for (i = 1; i <= 64; ++i)
        for (j = 1; j <= 64; ++j)
            A[i][j] = B[i][j];
}
```
**hiCUDA data directives**

```c
float A[66][66], B[66][66];
#pragma hicuda global alloc A[*][*] copyin
#pragma hicuda global alloc B[1:64][1:64]
for (iter = 0; iter < 10; ++iter)
{
    for (i = 1; i <= 64; ++i)
        for (j = 1; j <= 64; ++j)
    for (i = 1; i <= 64; ++i)
        for (j = 1; j <= 64; ++j)
            A[i][j] = B[i][j];
}
#pragma hicuda global copyout A[1:64][1:64]
```

**Evaluation of hiCUDA**

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

**hiCUDA 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
What to expect from the compiler

- Outlining of kernel regions
- Loop distribution and generation of guard code
- Redirection of accesses within kernel regions to corresponding GPU memory variables
  - Inter-procedural support

Performance Evaluation

- Generation of efficient (“cooperative”) code for data transfer to/from the shared memory
- Minimal allocation of shared memory to be used

Semantic verification
- All data needed by a kernel region are brought to the GPU memories
- Loop distribution does not break data dependences

Guidance to optimization
- Un-coalesced accesses to the global memory
- Bank conflicts of shared memory accesses

Hand-written CUDA (cuda) vs Compiler-generated CUDA

CUDA Benchmarks

MM CP SAD TPAF RPES
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 hiCUDA version was developed in 4 weeks
  - Both include the learning phase
- Disclaimer

Conclusions

- hiCUDA 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 hiCUDA 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 hiCUDA 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 hiCUDA directives

Related Work

- OpenMP to GPGPU (S. Lee, S.-J. Min, and R. Eigenmann)
  - Weak support in CUDA-specific features, like thread blocks and the shared memory
  - Many OpenMP directives are not necessary in data-parallel programming
- OpenCL
  - Involve similar “mundane” tasks as in CUDA
- PyCUDA (A. Klöckner)
  - A Python wrapper for CUDA
  - Requires programmers to write kernel functions explicitly
Related Work

- CUDA-lite (S. Ueng, M. Lathara, S. Baghsorkhi, W-M. Hwu)
  - Still requires the programmer to write CUDA code
  - Automation on an optimization pattern: utilizing the shard memory for coalescing global memory accesses

Thank You!