hiCUDA: A High-level Directive-based Language for GPU Programming

David Han

March 8, 2009
Outline

- Motivation of hiCUDA
- hiCUDA 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 (hiCUDA)

- 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. hiCUDA

<table>
<thead>
<tr>
<th>Typical CUDA programming steps</th>
<th>hiCUDA directives</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Identify and package a kernel</td>
<td>1. kernel</td>
</tr>
<tr>
<td>2. Partition kernel computation among a grid of GPU threads</td>
<td>2. loop_partition</td>
</tr>
<tr>
<td>3. Manage data transfer between the host memory and the GPU memory</td>
<td>3. global, constant</td>
</tr>
<tr>
<td>4. Perform memory optimizations</td>
<td>4. shared</td>
</tr>
</tbody>
</table>
An Example: Matrix Multiply

```c
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

```c
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;
    }
}
```
float A[32][96], B[96][64], C[32][64];
#pragma hicuda kernel matrixMul tblobc(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
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) {
}
CUDA Kernel for Matrix Multiplication

```c
float A[32][96], B[96][64], C[32][64];
#pragma hicuda kernel matrixMul tbloc(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
```
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

```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 tbblock(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

```c
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

```c
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 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
Performance Evaluation

![Bar chart showing normalized performance comparison between hand-written CUDA (cuda) and compiler-generated CUDA for different CUDA benchmarks.]

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

CUDA Benchmarks:
- MM
- CP
- SAD
- TPACF
- RPES

Normalized Performance

- Gear
- Hammer
- Screwdriver
- Spanner
- Wrench
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
Backup slides
Utilizing the shared memory

\[
\text{__shared__ } \mathbf{A}[\text{sharedA}[16][32]]
\]

\[
\text{__shared__ } \mathbf{B}[\text{sharedB}[32][16]]
\]

\[
\mathbf{A}[\text{sharedA}[kk+31][kk]]
\]

\[
\mathbf{B}[\text{sharedB}[kk][j]]
\]
Matrix Multiply Kernel in hiCUDA

#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 (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
            sum += A[i][k] * B[k][j];
        }
#pragma hicuda barrier
#pragma hicuda shared remove A B
    C[i][j] = sum;
}
}

#pragma hicuda kernel_end
Matrix Multiply Kernel in CUDA

```c
__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*bx + wB*ty + tx] = Csub;
}
```
Another configuration ...
Changes in *hiCUDA* code

```c
#pragma hicuda kernel matrixMul tblock(2,4) thread(16,16)
...
#pragma hicuda kernel_end

#pragma hicuda kernel matrixMul tblock(2,2) thread(16,32)
...
#pragma hicuda kernel_end
```
Changes in CUDA kernel code

```c
__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 = 32 * bx + wB * ty + tx, bStep = 32 * wB;

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

    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;
}
```
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

- **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
More Features of hiCUDA

- Support asynchronous kernel execution
  - nowait clause in the kernel directive

- Allow arbitrary dimensionality of the thread space

- Support BLOCK/CYCLIC distribution of loop iterations

- Support code execution by a single thread in each thread block
  - singular directive
More Features of *hiCUDa*

- Support the use of dynamic arrays in all data directives
  - *shape directive*

- Support allocation and transfer of array sections
  - `A[1:99][1:99]`

- Support data transfer across arrays with different names

- Support the use of constant memory
  - *constant directive*
OLD SLIDES
Kernel identification

```cpp
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
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
```
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
float A[32][96], B[96][64], C[32][64];
#pragma hicuda kernel matrixMult tblobk(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 A[32][96], B[96][64], C[32][64];
#pragma hin_cuda global alloc A[/*][/*] copyin
#pragma hin_cuda global alloc B[/*][/*] copyin
#pragma hin_cuda global alloc C[/*][/*]
#pragma hin_cuda kernel matrixMul tblock(2,4) thread(16,16)
#pragma hin_cuda loop_partition over_tblock over_thread
for (i = 0; i < 32; ++i) {
    #pragma hin_cuda 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 hin_cuda kernel_end
#pragma hin_cuda global copyout C[/*][/*]
#pragma hin_cuda global free A B C
Utilizing the shared memory

```c
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];
    }
    #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