A GPGPU Compiler for Memory Optimization and Parallelism Management

Yi Yang, Ping Xiang, Jingfei Kong, Huiyang Zhou

Department of Electrical and Computer Engineering
North Carolina State University

School of EECS
University of Central Florida
A Simplified View of GPU Architecture

- All processors run the same code, single program multiple data (SPMD)
- Communication among processors is costly.
Understanding GPU Architecture

• Processors are organized in groups, called Streaming Multiprocessors (SM).
• On-chip shared memory, a fast software-managed cache in each SM.
• Fast (local) communication among processors in a SM.
• Several memory controllers (MCs) shared among all the processors.
• Memory requests need to be evenly distributed among MCs. Otherwise, conflicts/partition clamping.
Thread Execution Model

- Threads are grouped in thread blocks (TB). Each TB is running on one streaming multiprocessor.
- 32 Threads in a thread block with consecutive thread ids form a **warp**, which is executed in the SIMD mode.
Key to Performance

- Bandwidth of global memory accesses
  - Coalesced global memory accesses
  - Distributed memory accesses among memory controllers/partitions

- Shared memory
  - Software-managed cache

- Balanced parallelism management: TLP vs. ILP
  - Thread level: register usage, ILP
  - Thread block level: shared memory usage, TLP
Outline

• Background
• Motivation
• Compiler Framework
  – Memory coalescing
  – Thread (block) merge
• Experimental Results
• Conclusion
Can we relieve the application developers of device specific optimizations?
Our Approach

Application developers

Algorithms

Naïve kernel

GPGPU compiler

High performance code

Handle the detailed GPGPU architecture

Identify fine-grain thread level parallelism

Global Memory

Shared Memory

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Proce

Identify fine-grain thread level parallelism

High performance code

GPGPU compiler

Naïve kernel

Algorithms

Application developers

Our Approach
Naïve Kernel

• Fine-grain data-level parallelism
• Compute one element/pixel in the output domain
• Example: Matrix multiplication

```c
float sum = 0;
for (int i=0; i<w; i++)
    sum += A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

Naïve matrix multiplication
Physical Meaning of the Naïve Kernel

• One thread computes one element at (idx, idy) in the product matrix

```c
float sum = 0;
for (int i=0; i<w; i++)
    sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

**Naïve matrix multiplication**
Outline

• Background
• Motivation
• Compiler Framework
  – Memory coalescing
  – Thread (block) merge
• Experimental Results
• Conclusion
Compiler Framework

Input: Naïve code

- Vectorization for memory access bandwidth
- Checking memory coalescing
- Converting non-coalesced accesses into coalesced
- Checking data dependencies and sharing patterns
- Thread & thread-block merge
- Data prefetching
- Removing memory partition camping

Output: Optimized performance code
Coalesced Global Memory Access

• Needed by GPU to achieve high memory bandwidth
• Examined at the half-warp granularity
• Requirements for coalesced global memory accesses
  – Aligned:
    • Half of warp threads must access the data with starting address to be a multiple of 64 bytes
  – Sequential (less strict for GTX 280/480):
    • Half of warp threads must access the data sequentially

<table>
<thead>
<tr>
<th>Thread</th>
<th>Global memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>128</td>
</tr>
<tr>
<td>15</td>
<td>188 192</td>
</tr>
</tbody>
</table>
Checking Memory Coalescing

float sum = 0;
for (int i=0; i<w; i++)
    sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;

Naïve matrix multiplication

A

B

i = 0
Checking Memory Coalescing

float sum = 0;
for (int i=0; i<w; i++)
    sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;

Naïve matrix multiplication
Checking Memory Coalescing

```
float sum = 0;
for (int i=0; i<w; i++)
    sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

**Naïve matrix multiplication**
Checking Memory Coalescing

float sum = 0;
for (int i=0; i<w; i++)
    sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;

**Naïve matrix multiplication**

A[idy](idx, idy)

16 threads

32 Threads in a thread block with consecutive thread ids form a **warp**, e.g., threads with id (idx, idy), (idx+1, idy),(idx+2, idy),..., (idx+31, idy) assuming idx is a multiple of 32.
Checking Memory Coalescing

```c
float sum = 0;
for (int i=0; i<w; i++)
    sum+=A[idy][i]*B[i][idx];
C[idy][idx] = sum;
```

**Naïve matrix multiplication**

32 Threads in a thread block with consecutive thread ids form a **warp**, e.g., threads with id (idx, idy), (idx+1, idy),(idx+2, idy),..., (idx+31, idy) assuming idx is a multiple of 32.
Checking Memory Coalescing

float sum = 0;
for (int i=0; i<w; i++)
    sum+=A[idy][i]*B[i][idx];
C[idy][idy] = sum;

Naïve matrix multiplication

C=AXB

i = 2
All 16 threads access one element A[idy][2].

32 Threads in a thread block with consecutive thread ids form a **warp**, e.g., threads with id (idx, idy), (idx+1, idy),(idx+2, idy),..., (idx+31, idy) assuming idx is a multiple of 32.
for (i=0; i<w; i=(i+16)) {
    __shared__ float shared0[16];
    shared0[(0+tidx)]=A[idy][(i+tidx)];
    __syncthreads();
    for (int k=0; k<16; k=(k+1)) {
        sum+=shared0[(0+k)]*B[(i+k)[idx]];
    }
    __syncthreads();
}

c[idy][idx] = sum;
Physical Meaning of the Coalesced Kernel

• One thread computes one element at (idx, idy) in the product matrix

• One thread block computes 16 elements in the product matrix

• Tile size: 16x1
Checking Data Dependence and Data Sharing

\[ C = AXB \]

\( (idx, idy) \)
Detect Data Sharing Among Thread Blocks

for (i=0; i<w; i=(i+16))
...A[idy][(i+tidx)]
tidx = 0 :15 as block size as 16 threads
i = 0
Thread blocks (0,0) – (0,3) access A[0][0:15]
Detect Data Sharing Among Thread Blocks

A[idy][(i+tidx)]

i = 16
Thread blocks (0,0) – (0,3) access A[0][16:31]
Detect Data Sharing Among Thread Blocks

\[
A[\text{idy}][(i+\text{tidx})]
\]

\(i = 32\)

Thread blocks (0,0) – (0,3) access \(A[0][32:47]\)

\[
A[\text{idy}][(i+\text{tidx})]
\]

Thread blocks in X direction access the same global memory
Thread Block Merge

• Preferred when shared data are in shared memory

Parallelism impact
• Increase the workload of each thread block
• Keep the workload of each thread

Improve memory reuse by merging neighboring thread blocks
```c
float sum = 0;
for (i=0; i<w; i=(i+16)) {
    __shared__ float shared0[16];
    if (tidx<16) {
        shared0[(0+tidx)]=a[idy][(i+tidx)+0]);
    }
    __syncthreads();
    int k;
    for (k=0; k<16; k=(k+1)) {
        sum+=shared0[(0+k)]*b[(i+k)][idx]);
    }
    __syncthreads();
}
```
```
c[idy][idx] = sum;
```

Thread block merge of MM
Physical Meaning of the Kernel (merged 2 blocks along the X direction)

- One thread computes one element at (idx, idy) in the product matrix
- One thread block computes 32 elements in the product matrix
- Tile size: 32x1
Thread Merge

- Preferred when shared data are in register file

Parallelism impact
- Increase thread workload (ILP)
- Keep the number of threads in a thread block

Improve memory reuse by merging threads from neighboring thread blocks.
float sum = 0;
for (i=0; i<w; i=(i+16)) {
  __shared__ float shared0[16];
  if (tidx<16) {
    shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
  }
  __syncthreads();
  int k;
  for (k=0; k<16; k=(k+1)) {
    sum+=shared0[(0+k)]*b[(i+k)][idx]);
  }
  __syncthreads();
}

c[idy][idx] = sum;
Code After Thread Merge

float sum_0 = 0;
......
float sum_31 = 0;
for (i=0; i<w; i=(i+16)) {
    __shared__ float shared0_0[16];
    ......
    __shared__ float shared0_31[16];
    if (tidx<16) {
        shared0_0[(0+tidx)]=
            a[idy*32+0][((i+tidx)+0)];
        ......
        shared0_31[(0+tidx)]=
            a[idy*32+31][((i+tidx)+0)];
    }
    __syncthreads();
}

int k;
for (k=0; k<16; k=(k+1)) {
    float r0 = b[(i+k)][idx]);
    sum_0+=shared0[(0+k)]*r0;
    ......
    sum_31+=shared0_31[(0+k)]*r0;
}
__syncthreads();
c[idy*32+0][idx] = sum_0;
......
c[idy*32+31][idx] = sum_31;
Physical Meaning of the Kernel (merged 2 threads along Y direction)

- One thread computes two element at (idx, idy) in the product matrix
- One thread block still has 16 threads (32 elements in the product matrix)
- Tile size: 16x2
Outline

• Background
• Motivation
• Compiler Framework
  – Memory coalescing
  – Thread (block) merge
• Experimental Results
• Conclusion
Experimental Methodology

• The proposed compiler is implemented as a source-to-source translator using Cetus [Lee et.al., LCPC 2003]

• Experimental environment
  – Operating system: 32-bit CentOS 5.2
  – CUDA SDK 2.2 for GTX8800 and GTX280
  – CUDA SDK 3.1 beta for GTX 480 (Fermi)

• 10 scientific/media processing algorithms
  – Naïve kernel code is available at http://code.google.com/p/gpgpucompiler/
Speedups over Naïve Kernels

Input: 4kx4k matrices or 4k vectors
Speedups over CUBLAS2.2 on GTX 280

- Similar performance for matrix multiplication, vector-vector multiplication, and reduction
- 14x-1.9x speedup for transpose matrix vector multiplication, matrix vector multiplication, and strsm
Summary

• We present an optimizing compiler for GPGPU programs.
• Our experimental results demonstrate the effectiveness of the proposed compiler optimizations.
• The open-source compiler website:
  – Contains the compiler code, the naïve kernels and the optimized kernels (optimized for GTX 280)

Thanks & Questions?
Impact of each optimization

### Speedups over naïve kernels

- **Coalescing**
- **Thread (block) merge**
- **Prefetching**
- **Partition camping elimination**

<table>
<thead>
<tr>
<th>Graphics Card</th>
<th>Coalescing</th>
<th>Thread (block) merge</th>
<th>Prefetching</th>
<th>Partition camping elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 8800</td>
<td>16</td>
<td>14</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>GTX 280</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>GTX 480</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- Coalesced memory access has less impact on the GTX 280 and GTX480.
- Thread (Block) merge achieve similar speedup over the code after the coalesced step (3.7x, 4.1x, 3.8x)
Partition Camping on Global Memory

• Global memory traffic should be evenly distributed among all the partitions

(a) Accesses to array A resulting in conflicts at partition 0

(b) Offset to eliminate the conflicts
Matrix vector multiplication on GTX 280

Opti_PC : the optimized kernel without partition camping elimination
Vectorization

Reduction on GTX280

Input: 1 million to 4 million complex numbers

<table>
<thead>
<tr>
<th>Performance (GFLOPS)</th>
<th>1M</th>
<th>2M</th>
<th>3M</th>
<th>4M</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimized wo vec</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>optimized</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>cublas</td>
<td>17</td>
<td>19</td>
<td>21</td>
<td>22</td>
</tr>
</tbody>
</table>

float a1 = A[2*idx];
float a2 = A[2*idx+1];

Before vectorization

float2* A_f2 = (float2*)A;
float2 a_f2 = A[2*idx];
float a1 = a_f2.x;
float a2 = a_f2.y;

After vectorization

43
CUBLAS on Fermi

The optimized version is based on GTX 280 configuration.
Transpose

Matrix Transpose

- gtx 480 diagonal
- gtx 480
- gtx 280 diagonal
- gtx 280
- gtx 8800 diagonal
- gtx 8800

Bar chart showing comparison of matrix transpose operations for different GPUs and matrix sizes (1kX1k, 2kX2k, 3kX3k, 4kX4k).
Speedups over naïve kernels

Input: 4k x 4k matrices or 4k vectors

128X
15X, 7.9X, 3.3X

Input:
- 4k x 4k matrices
- 4k vectors

Output:
- GPU performance comparisons (gtx8800, gtx280, gtx480)