Architecture-Aware Optimization Targeting Multithreaded Stream Computing

Byunghyun Jang, Synho Do, Homer Pien, and David Kaeli
Mar. 8 2009
Washington DC
Contents

▪ Motivations
▪ AMD RV670 Hardware and Brook+ Software
▪ Optimization Spaces
  – AMD's GPU Shader Analyzer
  – ALU, Texture Unit Utilization, and Thread Utilization
▪ Experimental Results
  – Matrix Multiplication
  – Back Projection of Medical Image Reconstruction
▪ Conclusion and Future Work
Motivations

- GPU is the platform of choice for compute intensive data parallel applications but ...

- Optimization of GPGPU is a very challenging task
  - Time consuming (trial and error method)
  - Hampered by lack of details about underlying hardware
Motivations

▪ GPU is the platform of choice for compute intensive data parallel applications but ...

▪ Optimization of GPGPU is a very challenging task
  – Time consuming (trial and error method)
  – Hampered by lack of details about underlying hardware

▪ AMD's stream computing has different hardware architecture, programming model from NVIDIA CUDA, resulting different optimization spaces

▪ AMD's recent announcement of architectural details (e.g. ISA) and profiling tool at low level has allowed for aggressive optimizations
AMD RV670 Hardware

- Mainstream in AMD's stream computing lineup
- First GPU to support double precision for GPGPU
- 320 (64x5) stream processors
- 64 enhanced transcendental units (COS, LOG, EXP, etc.)
- ~75 GB/s memory bandwidth
- Peak processing capabilities of 497 GFLOPs
- Targeted for handling thousands of simultaneous lightweight threads

excerpted from SIGGRAPH 2008 presentation
AMD RV670 Hardware

- Mainstream in AMD's stream computing lineup
- First GPU to support double precision for GPGPU
- 320 (64x5) stream processors
- 64 enhanced transcendental units (COS, LOG, EXP, etc.)
- ~75 GB/s memory bandwidth
- Peak processing capabilities of 497 GFLOPs
- Targeted for handling thousands of simultaneous lightweight threads
Brook+ Programming Software

- Extension to C language and based on BrookGPU
- Stream programming model
  - Co-processing model
  - Input, output streams
  - Domain of execution
  - Allow programming at intermediate language (IL) level

Excerpted from AMD's web material
Optimization Spaces

- Peak performance is achieved when no processor is idle while minimizing the number of redundant instructions.

- Optimization is broken down to 3 spaces, based on its contribution to GPU performance and our analysis of contention points:
  - **ALU** Unit Utilization
  - **Texture** Unit Utilization
  - **Thread** Utilization

- **GPU Shader Analyzer** is used to investigate machine binaries and its statistics.
AMD GPU Shader Analyzer

Northeastern University Computer Architecture Research Group
AMD GPU Shader Analyzer

High Level Kernel Source
AMD GPU Shader Analyzer

Compiled Assembly Code

Northeastern University Computer Architecture Research Group
ALU Unit Utilization

- A shader processing unit (SPU) is a 5 way scalar VLIW processor
- Utilizing all of these ALU units efficiently (high VLIW slot occupancy and reduced ALU instruction count) is a key
- The compiler plays a critical role but is restricted by programming style present in original program
- We observed two efficient techniques in this context
  - Use of intrinsic function, merging subfunctions whenever possible
ALU Unit Utilization Case 1

Use of Intrinsic Function

(a) before using intrinsic function

\[
t5 = t12.x \times vx.x + t12.y \times vx.y; \\
t6 = t12 \times vy.x + t12.y \times vy.y;
\]

(b) after using intrinsic dot function

\[
t5 = \text{dot} (t12, vx); \\
t6 = \text{dot} (t12, vy);
\]
**ALU Unit Utilization Case 1**

Use of Intrinsic Function

(a) before using intrinsic function

\[
\begin{align*}
    t5 &= t12.x \times vx.x + t12.y \times vx.y; \\
    t6 &= t12 \times vy.x + t12.y \times vy.y;
\end{align*}
\]

(b) after using intrinsic `dot` function

\[
\begin{align*}
    t5 &= \text{dot}(t12, vx); \\
    t6 &= \text{dot}(t12, vy);
\end{align*}
\]

Compile

22  y: MUL_e ___, PS21, R4.y  
    t: COS ___, T1.x
23  x: ADD R3.x, T0.w, PV22.y  
    w: MUL_e R0.w, -PS22, T1.z  
    t: SIN R3.w, T0.z

Compile

22  y: MULADD_e R3.y, R2.x, ...  
    t: COS ___
23  x: MULADD_e R3.z, R0.w -PS22, T1.z  
    t: SIN R3.2, T0.w

Total instruction counts:

```
       Less
```

Northeastern University Computer Architecture Research Group
ALU Unit Utilization Case 2

Merging subfunctions

(a) before merging subfunctions

\[ j_1 = \text{locpix1} (\text{Beta}_i, \text{numChn}, \text{BD}, \text{numProj}); \]
\[ j_2 = \text{locpix1} (\text{Z}_i, \text{numRow}, \text{ZD}, \text{numProj}); \]

(b) after merging subfunctions

\[ j = \text{locpix} (\text{Beta}_i, \text{Z}_i, \text{numChn}, \text{numRow}, \text{BD}, \text{ZD}, \text{numProj}); \]
ALU Unit Utilization Case 2

Merging subfunctions

\[ j_1 = \text{locpix1}(\text{Beta}_i, \text{numChn}, \text{BD}, \text{numProj}); \]
\[ j_2 = \text{locpix1}(Z_i, \text{numRow}, ZD, \text{numProj}); \]

(a) before merging subfunction

\[ j = \text{locpix}(\text{Beta}_i, \text{Zi}, \text{numChn}, \text{numRow}, \text{BD}, ZD, \text{numProj}); \]

(b) after merging subfunctions

Compiled with

Less total ALU instruction counts
Texture Unit Utilization

- A GPU is designed to provide high memory bandwidth rather than low memory latency.
- Efficient texture unit utilization is critical to get near peak computing power.
- Unlike CUDA, Brook+ provides built-in short vector types to allow code to be explicitly tuned for available SIMD machine.
- We propose two techniques here:
  - Use of vector type, multiple output streams.
Texture Unit Utilization Case 1

Vector Type

(a) before using vector type

float tmpA = A[indexA.xy];
float tmpB = B[indexB.xy];

(b) after using vector type

float4 A11 = A1[index.wy];
float4 B11 = B1[index.xw];
float4 B22 = B2[index.xw];
float4 B33 = B3[index.xw];
float4 B44 = B4[index.xw];

compile

04 TEX: ADDR(64) CNT(2) VALID_PIX
 8 SAMPLE_LZ R0.x__, ...
 9 SAMPLE_LZ R1.x__, ...

compile

04 TEX: ADDR(112) CNT(5) VALID_PIX
 9 SAMPLE_LZ R4.xyzw, ...
10 SAMPLE_LZ R0.xyzw, ...
11 SAMPLE_LZ R3.xyzw, ...
12 SAMPLE_LZ R1.xyzw, ...
13 SAMPLE_LZ R2.xyzw, ...
Texture Unit Utilization Case 1

Vector Type

(a) before using vector type

float tmpA = A[indexA.xy];
float tmpB = B[indexB.xy];

(b) after using vector type

float4 A11 = A1[index.wy];
float4 B11 = B1[index.xw];
float4 B22 = B2[index.xw];
float4 B33 = B3[index.xw];
float4 B44 = B4[index.xw];
Texture Unit Utilization Case 2

Multiple Output Streams

(a) one output stream

(b) multiple output streams
Texture Unit Utilization Case 2

Multiple Output Streams

(a) single output stream

(b) multiple output streams

5 texture instructions in a clause
5/8 = 63%

8 texture instructions in a clause
8/8 = 100%
Thread Utilization

- Necessary conditions for maximal thread utilization
  - Large number of total threads to fully exploit the power of GPU's parallel resources
Thread Utilization

- **Necessary conditions for maximal thread utilization**
  - Large number of total threads to fully exploit the power of GPU's parallel resources
Necessary conditions for maximal thread utilization

- Large number of total threads to fully exploit the power of GPU's parallel resources
- High arithmetic intensity (ALU to texture ratio)
Thread Utilization

- Necessary conditions for maximal thread utilization
  - Large number of total threads to fully exploit the power of GPU's parallel resources
  - High arithmetic intensity (ALU to texture ratio)
  - Large number of active threads are necessary to hide memory access latencies
    - Limited hardware resources – general purpose register
Thread Utilization

- Necessary conditions for maximal thread utilization
  - Large number of total threads to fully exploit the power of GPU's parallel resources
  - High arithmetic intensity (ALU to texture ratio)
  - Large number of active threads are necessary to hide memory access latencies
    - Limited hardware resources – general purpose register
- Merging scalar variables into vector whenever possible
Thread Utilization

- Necessary conditions for maximal thread utilization
  - Large number of total threads to fully exploit the power of GPU's parallel resources
  - High arithmetic intensity (ALU to texture ratio)
  - Large number of active threads are necessary to hide memory access latencies
    - Limited hardware resources – general purpose register
- Merging scalar variables into vector whenever possible
- Loop unrolling
Experimental Results

- Experimental settings
  - AMD's stream SDK 1.1 beta with Catalyst 8.6 display driver
  - AMD Radeon HD 3870 (RV670)
  - Intel Core 2 duo (2.66 GHz, 2GB main memory)
Experimental Results

- **Experimental settings**
  - AMD's stream SDK 1.1 beta with Catalyst 8.6 display driver
  - AMD Radeon HD 3870 (RV670)
  - Intel Core 2 duo (2.66 GHz, 2GB main memory)
  - Two benchmarks
    - Matrix multiplication
    - Back projection from medical image reconstruction
    - From naïve implementation to more optimized ones
Experimental Results - Matrix Multiplication

### Optimizations

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>ALU</th>
<th>Texture</th>
<th>Arithmetic Intensity</th>
<th>GPR</th>
<th>Speedup over Naive</th>
<th>Speedup over CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Util.</td>
<td>Count</td>
<td>Util.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>12</td>
<td>50%</td>
<td>2</td>
<td>25%</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Variable Reduction</td>
<td>11</td>
<td>52.7%</td>
<td>2</td>
<td>25%</td>
<td>5.5</td>
<td>7</td>
</tr>
<tr>
<td>Loop Unrolling (4x)</td>
<td>20</td>
<td>71%</td>
<td>8</td>
<td>25%</td>
<td>2.5</td>
<td>16</td>
</tr>
<tr>
<td>Loop Unrolling (8x)</td>
<td>30</td>
<td>84.67%</td>
<td>16</td>
<td>25%</td>
<td>1.88</td>
<td>17</td>
</tr>
<tr>
<td>Loop Unrolling (16x)</td>
<td>52</td>
<td>91.92%</td>
<td>32</td>
<td>25%</td>
<td>1.63</td>
<td>25</td>
</tr>
<tr>
<td>Vector Type</td>
<td>17</td>
<td>78.82%</td>
<td>5</td>
<td>100%</td>
<td>3.4</td>
<td>9</td>
</tr>
<tr>
<td>Multiple Output</td>
<td>78</td>
<td>93.59%</td>
<td>12</td>
<td>100%</td>
<td>6.5</td>
<td>34</td>
</tr>
</tbody>
</table>
Experimental Results - Matrix Multiplication

Optimizations | ALU Count | Util. % | Texture Count | Util. % | Arithmetic Intensity | GPR | Speedup over Naive | Speedup over CPU
---|---|---|---|---|---|---|---|---
Naive | 12 | 50% | 2 | 25% | 6 | 9 | 1 | 131.76
Variable Reduction | 11 | 52.7% | 2 | 25% | 5.5 | 7 | 1.01 | 132.07
Loop Unrolling (4x) | 20 | 71% | 8 | 25% | 2.5 | 16 | 1.05 | 138.91
Loop Unrolling (8x) | 30 | 84.67% | 16 | 25% | 1.88 | 17 | 1.09 | 143.58
Loop Unrolling (16x) | 52 | 91.92% | 32 | 25% | 1.63 | 25 | 1.07 | 141.47
Vector Type | 17 | 78.82% | 5 | 100% | 3.4 | 9 | 4.78 | 629.64
Multiple Output | 78 | 93.59% | 12 | 100% | 6.5 | 34 | 6.7 | 882.94
Experimental Results - Matrix Multiplication

![Graph showing performance results for different matrix sizes and optimizations.](image)

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>ALU</th>
<th>Texture</th>
<th>Arithmetic Intensity</th>
<th>GPR</th>
<th>Speedup over Naive</th>
<th>Speedup over CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Util.</td>
<td>Count</td>
<td>Util.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>12</td>
<td>50%</td>
<td>2</td>
<td>25%</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Variable Reduction</td>
<td>11</td>
<td>52.7%</td>
<td>2</td>
<td>25%</td>
<td>5.5</td>
<td>7</td>
</tr>
<tr>
<td>Loop Unrolling (4x)</td>
<td>20</td>
<td>71%</td>
<td>8</td>
<td>25%</td>
<td>2.5</td>
<td>16</td>
</tr>
<tr>
<td>Loop Unrolling (8x)</td>
<td>30</td>
<td>84.67%</td>
<td>16</td>
<td>25%</td>
<td>1.88</td>
<td>17</td>
</tr>
<tr>
<td>Loop Unrolling (16x)</td>
<td>52</td>
<td>91.92%</td>
<td>32</td>
<td>25%</td>
<td>1.63</td>
<td>25</td>
</tr>
<tr>
<td>Vector Type</td>
<td>17</td>
<td>78.82%</td>
<td>5</td>
<td>100%</td>
<td>3.4</td>
<td>9</td>
</tr>
<tr>
<td>Multiple Output</td>
<td>78</td>
<td>93.59%</td>
<td>12</td>
<td>100%</td>
<td>6.5</td>
<td>34</td>
</tr>
</tbody>
</table>

Northeastern University Computer Architecture Research Group
Experimental Results - Matrix Multiplication

Northeastern University Computer Architecture Research Group
Experimental Results - Matrix Multiplication

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>ALU</th>
<th>Texture</th>
<th>Arithmetic Intensity</th>
<th>GPR</th>
<th>Speedup over</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Util.</td>
<td>Count</td>
<td>Util.</td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>12</td>
<td>50%</td>
<td>2</td>
<td>25%</td>
<td>6</td>
</tr>
<tr>
<td>Variable Reduction</td>
<td>11</td>
<td>52.7%</td>
<td>2</td>
<td>25%</td>
<td>5.5</td>
</tr>
<tr>
<td>Loop Unrolling (4x)</td>
<td>20</td>
<td>71%</td>
<td>8</td>
<td>25%</td>
<td>2.5</td>
</tr>
<tr>
<td>Loop Unrolling (8x)</td>
<td>30</td>
<td>84.67%</td>
<td>16</td>
<td>25%</td>
<td>1.88</td>
</tr>
<tr>
<td>Loop Unrolling (16x)</td>
<td>52</td>
<td>91.92%</td>
<td>32</td>
<td>25%</td>
<td>1.63</td>
</tr>
<tr>
<td><strong>Vector Type</strong></td>
<td>17</td>
<td>78.82%</td>
<td>5</td>
<td>100%</td>
<td>3.4</td>
</tr>
<tr>
<td>Multiple Output</td>
<td>78</td>
<td>93.59%</td>
<td>12</td>
<td>100%</td>
<td>6.5</td>
</tr>
</tbody>
</table>

Northeastern University Computer Architecture Research Group
Experimental Results - Matrix Multiplication

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>ALU</th>
<th>Texture</th>
<th>Arithmetic Intensity</th>
<th>GPR</th>
<th>Speedup over Naive</th>
<th>Speedup over CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>12 50%</td>
<td>2 25%</td>
<td>6</td>
<td>9</td>
<td>1.01</td>
<td>131.76</td>
</tr>
<tr>
<td>Variable Reduction</td>
<td>11 52.7%</td>
<td>2 25%</td>
<td>5.5</td>
<td>7</td>
<td>1.65</td>
<td>138.91</td>
</tr>
<tr>
<td>Loop Unrolling (4x)</td>
<td>20 71%</td>
<td>8 25%</td>
<td>2.5</td>
<td>16</td>
<td>1.69</td>
<td>143.58</td>
</tr>
<tr>
<td>Loop Unrolling (8x)</td>
<td>30 84.67%</td>
<td>16 25%</td>
<td>1.88</td>
<td>17</td>
<td>1.07</td>
<td>141.47</td>
</tr>
<tr>
<td>Loop Unrolling (16x)</td>
<td>52 91.92%</td>
<td>32 25%</td>
<td>1.63</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Type</td>
<td>17 78.82%</td>
<td>5 100%</td>
<td>3.4</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple Output</td>
<td>78 93.59%</td>
<td>12 100%</td>
<td>6.5</td>
<td>34</td>
<td>6.7</td>
<td>882.94</td>
</tr>
</tbody>
</table>
Experimental Results - Back Projection

- Limitations of Stream Programming Model
- Rich ALU optimizations spaces
- Inefficient texture optimizations
  - Dimensions of input and output streams Mismatch
  - Unable to apply multiple output stream due to too large kernel body

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>ALU</th>
<th>Texture</th>
<th>Arithmetic Intensity</th>
<th>GPR</th>
<th>Speedup over Naive</th>
<th>Speedup over CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cnt.</td>
<td>Util.</td>
<td>Cnt.</td>
<td>Util.</td>
<td></td>
<td>I1</td>
</tr>
<tr>
<td>Naive</td>
<td>112</td>
<td>42.7%</td>
<td>24</td>
<td>25%</td>
<td>4.67</td>
<td>26</td>
</tr>
<tr>
<td>ALU Unit Util.</td>
<td>98</td>
<td>45.9%</td>
<td>24</td>
<td>25%</td>
<td>4.08</td>
<td>15</td>
</tr>
<tr>
<td>Vector Type (input)</td>
<td>385</td>
<td>45.7%</td>
<td>66</td>
<td>30.7%</td>
<td>5.83</td>
<td>55</td>
</tr>
<tr>
<td>Vector Type (output)</td>
<td>205</td>
<td>42.0%</td>
<td>44</td>
<td>25%</td>
<td>4.66</td>
<td>38</td>
</tr>
</tbody>
</table>
Experimental Results - Back Projection

- Limitations of Stream Programming Model
- Rich ALU optimizations spaces
- Inefficient texture optimizations
  - Dimensions of input and output streams Mismatch
  - Unable to apply multiple output stream due to too large kernel body

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>ALU</th>
<th>Texture</th>
<th>Arithmetic Intensity</th>
<th>GPR</th>
<th>Speedup over Serial C version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>112</td>
<td>42.7%</td>
<td>24</td>
<td>25%</td>
<td>4.67</td>
</tr>
<tr>
<td>ALU Unit Util.</td>
<td>98</td>
<td>45.9%</td>
<td>24</td>
<td>25%</td>
<td>4.08</td>
</tr>
<tr>
<td>Vector Type (input)</td>
<td>385</td>
<td>45.7%</td>
<td>66</td>
<td>30.7%</td>
<td>5.83</td>
</tr>
<tr>
<td>Vector Type (output)</td>
<td>205</td>
<td>42.0%</td>
<td>44</td>
<td>25%</td>
<td>4.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental Results - Back Projection

- Rich ALU optimizations spaces
- Restrictions in Stream Programming Model
- Inefficient texture optimizations
  - Dimensions of input and output streams Mismatch
  - Unable to apply multiple output stream due to too large kernel body

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>ALU</th>
<th>Texture</th>
<th>Arithmetic Intensity</th>
<th>GPR</th>
<th>Speedup over Naive</th>
<th>Speedup over CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cnt.</td>
<td>Util.</td>
<td>Cnt.</td>
<td>Util.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive</td>
<td>112</td>
<td>42.7%</td>
<td>24</td>
<td>25%</td>
<td>4.67</td>
<td>26</td>
</tr>
<tr>
<td>ALU Unit Util.</td>
<td>98</td>
<td>45.9%</td>
<td>24</td>
<td>25%</td>
<td>4.08</td>
<td>15</td>
</tr>
<tr>
<td>Vector Type (input)</td>
<td>385</td>
<td>45.7%</td>
<td>66</td>
<td>30.7%</td>
<td>5.83</td>
<td>55</td>
</tr>
<tr>
<td>Vector Type (output)</td>
<td>205</td>
<td>42.0%</td>
<td>44</td>
<td>25%</td>
<td>4.66</td>
<td>38</td>
</tr>
</tbody>
</table>
Experimental Results - Back Projection

Image reconstructed on CPU

Image reconstructed on GPU
Conclusion and Future Work

- Optimization has been a major hurdle in GPGPU research field.
- An **efficient and structured optimization methodology** by inspecting disassembled machine code in an aid of vendor's profiling tool is proposed.
- We demonstrated its effectiveness using two benchmarks which show different optimization spaces.
- Our future work includes memory access pattern based approach and optimizations on IL level programming.
Questions ?
Thank you!