Understanding the Performance of GPGPU Applications from a Data-Centric View

Hui Zhang  w.hzhang86@samsung.com
Jeffrey K. Hollingsworth  hollings@umd.edu
Motivation

- It’s hard for programmers to write efficient code on highly parallel and heterogeneous architectures.

- There are few performance tools for CUDA users that can locate inefficient source code and guide user-level optimizations.

- Traditional Code-centric profiling approach is insufficient in investigating data placement issue.
Contributions

• First, the tool offers **fine-grained**, in-depth performance analysis into the kernel execution, providing programmers with finer insight into the GPU kernel execution.

• Second, the tool uses a **data-centric** performance analysis technique to map performance data back to variables in the source code.

• Third, it proposes a method to get the **complete calling context** profiling, including the CPU call stack before a kernel is launched and the GPU call stack within a kernel.
CUDA Programming Overview

```c
int busy(int *x) {
    // hotspot function
    *x = complex();
    return *x;
}

int main() {
    for (i=0; i<n; i++) {
        A[i] = busy(&B[i]) +
               busy(&C[i-1]) +
               busy(&C[i+1]);
    }
}
```

**Data-centric Profiling**

- `main`: 100%
- `busy`: 100%
- `complex`: 100%

**Code-centric Profiling**

- `A`: 100%
- `B`: 33.3%
- `C`: 66.7%
"I didn’t say you were to blame... I said I am blaming you."

Properly Assign Blame
CUDABlamer Framework

Static Analysis
- Data flow analysis
- Control flow analysis
- Intra-procedural Blame analysis
- Exit variables analysis

Monitored Execution
- CUPTI Callback API: tag kernel invocation
- Libunwind: CPU stack unwinding
- CUPTI Activity API: GPU kernel sampling

Postmortem Process
- Process runtime information
- Reconstruct CPU&GPU calling context
- Inter-procedural Blame analysis
- Determine Blame attribution vars/funcs

GUI Presentation
- Data-centric profiling result
- Code-centric profiling result
CUDABlamer – Static Analysis

- Graphical Representation to resolve Blame relation

\[
\begin{align*}
\text{var } a & \text{ : int = 6;} \\
\text{var } b & \text{ : int = 7;} \\
\text{var } c & \text{ : int = a + b;}
\end{align*}
\]

- Resolve LLVM composite instructions to propagate blame hierarchically

\[
\%\text{id}x_1 = \text{getelementptr i32, } \%\text{struct.MyStruct* } \%\text{myVar, i64 0, i64 0}
\]

(a) Normal GEP instruction

\[
\%\text{id}x_2 = \text{getelementptr i32, } \{\text{getelementptr i32*, } \\
\%\text{struct.MyStruct* } \%\text{myVar, i64 0, i64 1}, \text{i64 0}
\]

(b) Composite GEP instruction
CUDABlamer – Postmortem Process

- Construct Calling Context for CPU-GPU Hybrid Model
  - CPU stack: keep call stack with *Kernel Launch ID (correlationID)*
  - GPU stack for kernel execution: find all paths from sample point to kernel using Depth-First-Search (top & bottom node info from ActivityAPI)
  - Reconstruct the calling context: Connect CPU & GPU stacks through *correlationID*

```c
__global__ void kernelFunc(...){
    foo(); ... 
    bar(); ... 
}

__device__ void foo() {
    bar(); ... 
    x = 1; ... //Sample 1 
    y = 2; ... //Sample 2 
}

__device__ void bar() {
    A[i] = B[i]*s; //Sample 3 
}
```

Example

Ambiguity: 2 possible call paths from the sample point to "kernelFunc"
Precision Evaluation

- Coverage Metric:

\[
\text{coverage} = \frac{\text{totalNumSamples} - \text{numAmbiSamples}}{\text{totalNumSamples}}
\]
Tool Evaluation – Particlefilter

Single-node: 2 NVIDIA Tesla P100 GPUs, each P100 GPU contains 16 GB on-chip memory and 56 SM (streaming multiprocessors). Each SM also has 64KB of shared memory. The GPU also provides 48KB of constant memory.

Compilers: nvcc 8.0, gcc 4.8.5 and clang 4.0.1

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Context</th>
<th>Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>ye/xe</td>
<td>double</td>
<td>main.particleFilter</td>
<td>100%</td>
</tr>
<tr>
<td>arrayX/arrayY</td>
<td>*double</td>
<td>main.particleFilter</td>
<td>100%</td>
</tr>
<tr>
<td>xj</td>
<td>*double</td>
<td>main.particleFilter</td>
<td>97.9%</td>
</tr>
<tr>
<td>yj</td>
<td>*double</td>
<td>main.particleFilter</td>
<td>97.8%</td>
</tr>
<tr>
<td>xj_GPU</td>
<td>*double</td>
<td>main.particleFilter</td>
<td>97.9%</td>
</tr>
<tr>
<td>yj_GPU</td>
<td>*double</td>
<td>main.particleFilter</td>
<td>97.8%</td>
</tr>
<tr>
<td>index</td>
<td>int</td>
<td>main.particleFilter.kernel</td>
<td>95.7%</td>
</tr>
</tbody>
</table>
Tool Evaluation – Particlefilter

• **Optimization**
  - using constant memory for read-only variables `arrayX_GPU`, `arrayY_GPU`, `u_GPU`, `CDF_GPU`
Tool Evaluation - Gesummv

- Gesummv is part of the Polybench test suite and has a kernel that does scalar, vector, and matrix multiplication.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Context</th>
<th>Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>y_outputFromGpu</td>
<td>*float</td>
<td>main</td>
<td>100%</td>
</tr>
<tr>
<td>y_gpu</td>
<td>*float</td>
<td>main.gesummvCuda</td>
<td>100%</td>
</tr>
<tr>
<td>tmp_gpu</td>
<td>*float</td>
<td>main.gesummvCuda</td>
<td>52.1%</td>
</tr>
<tr>
<td>j</td>
<td>int</td>
<td>gesummv_kernel</td>
<td>4.3%</td>
</tr>
<tr>
<td>A_gpu/B_gpu</td>
<td>*float</td>
<td>main.gesummvCuda</td>
<td>1.2%</td>
</tr>
<tr>
<td>x_gpu</td>
<td>*float</td>
<td>main.gesummvCuda</td>
<td>1.2%</td>
</tr>
</tbody>
</table>
Tool Evaluation - Gesummv

- **Optimization**
  - $y_{gpu}$ is allocated in the global memory and updating it iteratively is costly. We use temporary variables to hold intermediate result in the for loop and assigning the ultimate value to the corresponding array element once in the end.
Tool Evaluation - Gramschm

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Context</th>
<th>Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_outputFromGpu</td>
<td>*float</td>
<td>main</td>
<td>99.1%</td>
</tr>
<tr>
<td>A_gpu</td>
<td>*float</td>
<td>main.gramschmidtCuda</td>
<td>99.1%</td>
</tr>
<tr>
<td>R_gpu</td>
<td>*float</td>
<td>main.gramschmidtCuda</td>
<td>60.6%</td>
</tr>
<tr>
<td>nrm</td>
<td>float</td>
<td>main.gramschmidtCuda</td>
<td>19.5%</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>Gramschmidt_kernel3</td>
<td>6.7%</td>
</tr>
<tr>
<td>Q_gpu</td>
<td>*float</td>
<td>main.gramschmidtCuda</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Scope</th>
<th>Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>CPU</td>
<td>100%</td>
</tr>
<tr>
<td>gramschmidtCuda</td>
<td>CPU</td>
<td>100%</td>
</tr>
<tr>
<td>gramschmidt_kernel3</td>
<td>GPU</td>
<td>78.2%</td>
</tr>
<tr>
<td>gramschmidt_kernel1</td>
<td>GPU</td>
<td>19.9%</td>
</tr>
<tr>
<td>gramschmidt_kernel2</td>
<td>GPU</td>
<td>1.9%</td>
</tr>
</tbody>
</table>
Tool Evaluation - Gramschm

- **Optimization**
  - $R_{gpu}$: Use a temporary variable to hold the incremental value of $R_{gpu}$ and do one-time assignment after the loop
  - $Q_{gpu}$: Use shared memory instead of global memory to store per-block copy of it, and change the column-based access to row-based access

![Gramschm Diagram]

- Original Kernel Execution Time (ms): 2.89
- Optimized Kernel Execution Time (ms): 0.51
- Speedup: 5.7x
# CUDABlamer Overhead

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Clean execution</th>
<th>Static analysis</th>
<th>Monitored execution</th>
<th>Post processing</th>
<th>Runtime overhead</th>
<th>Total overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotspot</td>
<td>10.43</td>
<td>1.61</td>
<td>10.82</td>
<td>0.83</td>
<td>3.7%</td>
<td>27.0%</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>16.96</td>
<td>2.54</td>
<td>115.35</td>
<td>55.46</td>
<td>580%</td>
<td>922%</td>
</tr>
<tr>
<td>Particlefilter</td>
<td>10.21</td>
<td>1.34</td>
<td>11.1</td>
<td>1.74</td>
<td>8.7%</td>
<td>38.9%</td>
</tr>
</tbody>
</table>

Unit: seconds

- Static analysis runs once for each benchmark w/ different problem sizes
- Post processing overhead depends on #samples & #blame variables/sample
- Runtime overhead = (Monitored execution / Clean execution) - 1
- Total overhead = (Total profiling time / Clean execution) - 1

- **Runtime overhead** can be high due to the poor performance of CUPTI library provided by NVIDIA when using `PC_SAMPLING` mechanism
Conclusion

• New Performance Attribution for Emerging Programming Models
  o Developed a data-centric CUDA profiler: CUDABlamer

• Complete User-level Calling Context
  o Using static and runtime information to interpolate the complete calling context for heterogeneous architecture

• Valuable Performance Insights
  o Manual optimization gained speedup up to 47x for selected CUDA kernels