A Script-Based Autotuning Compiler System to Generate High-Performance CUDA code

Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, Jacqueline Chame
Motivation

Challenges to programming the GPU

- Parallel Computation Partitioning
- Data placement in memory hierarchy
- Memory Bandwidth optimizations

Best solution depends on architecture and input data set
## Target Devices

<table>
<thead>
<tr>
<th></th>
<th>GTX-280</th>
<th>C2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>#SMs</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td>Cores/SM</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Total cores</td>
<td>240</td>
<td>448</td>
</tr>
<tr>
<td>Peak (SP)</td>
<td>933 GF/s</td>
<td>1.03 TF/s</td>
</tr>
<tr>
<td>Peak (DP)</td>
<td>87 GF/s</td>
<td>515 GF/s</td>
</tr>
<tr>
<td>Global memory</td>
<td>1 GB</td>
<td>3 GB</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>142 GB/s</td>
<td>144 GB/s</td>
</tr>
<tr>
<td>Shared memory/ SM</td>
<td>16 KB</td>
<td>(up to) 48 KB</td>
</tr>
<tr>
<td>Registers/ SM</td>
<td>16K</td>
<td>32K</td>
</tr>
<tr>
<td>“Texture” accesses</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Data cache</td>
<td>0</td>
<td>(up to) 48 KB</td>
</tr>
</tbody>
</table>

C2050: more cores / SM, but fewer SMs

C2050: less registers and shared memory per core
Our Approach

Autotuning

• Automatically generate a “search space” of possible implementations
• A code variant represents a unique implementation amongst many
• A parameter represents a discrete set of values that govern code generation of a variant

A layered autotuning compiler framework

• Enhances productivity of naïve and expert programmers
• Separates code generation from optimization strategy
• Facilitates exploration of a search space of strategies
Compiler Framework for Autotuning

Key Idea in the paper
Transformation Strategy Generator

Sequential C code

Optimization strategies (recipes)

Transformation Strategy Decisions

Autotuning

CUDA-CHiLL

CHiLL

High Performance CUDA code
TSG: Four Phase Approach

Phase I: Identify Candidate Computation Decompositions
Phase II: Data Placement Candidates in Memory Hierarchy
Phase III: Select computation decomposition and data placement
Phase IV: Parameter Autotuning and Code Generation

Generating the search space
Pruning the search space and Autotuning
TSG: Identifying Strategy

Parallel Mapping:
- Constrained by dependences
- Global memory coalescing in x thread dimension
- Evaluate data partitioning options

Manage Heterogeneous Memory Hierarchy:
- **Register Candidate**: Reuse in a thread
- **Shared Memory**: Reuse across thread or non-coalesced global memory access
- **Texture Memory**: Read-only data already mapped to registers or shared memory
- **Constant Memory**: Read-only, with reuse across threads (and blocks) and short reuse distance
Example: Matrix-Matrix Multiply

for l
for j
for k
    c[j][i] += a[k][i] * b[j][k];

Data placement candidates:
- Registers: c
- Shared memory: a, b
- Texture memory: a, b
- Global memory: a
- Constant memory: <empty>

Data partitioning options
- C: cyclic x, block y
- C: cyclic x, cyclic y

I and j loops: tile to create block and thread dimensions
2-d data structures → 2-d blocks and threads

Dependence carried by k loop
Move inside thread

Global memory coalescing – i loop as x thread dimension
Example: Matrix-Matrix Multiply

TC2050 Fermi implementation
Mostly corresponds to CUBLAS 3.2 and MAGMA

GTX-280 implementation
Mostly corresponds to CUBLAS 2.x and Volkov’s SC08 paper

Different computation decomposition leads to additional tile command
Now a also in (larger) shared memory, both a and b are read through texture memory

```
1 tile_by_index\{"i","j\},\{TI,TJ\},\{l1_control="ii",l2_control="jj"\},
    \{"ii","jj","i","j\}
2 tile_by_index\{"k\},\{TK\},\{l1_control="kk"\},
    \{"ii","jj","kk","i","j","k\},\text{strided}
3 tile_by_index\{"i\},\{TK\},\{l1_control="tt\},l1_tile="t",
    \{"ii","jj","kk","tt","t","j","k\}
4 cudaize("mm_GPU",\{a=N*N,b=N*N,c=N*N\},
    \{block=\{"ii","jj\},\text{thread=\{"t","tt\}"}\}
5 copy_to_shared("tx","b",-16)
6 copy_to_registers("kk","c")
7 copy_to_texture("b")
8 unroll_to_depth(2)
```
Pruning the Search Space

**Parameterized Code Variant Selection**
- Heuristics limit variants to promising ones
- Based on architecture features and data access properties (coalescing, reuse within/across threads)

**Optimization Parameters**
- Architectural knowledge (warp size, memory capacities)
- Load balance model

TSG autotunes parameterized variant selection and optimization parameters independently to further help prune the search space
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Domain</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>Linear Algebra</td>
<td>CUBLAS</td>
</tr>
<tr>
<td>MV</td>
<td>Linear Algebra</td>
<td>CUBLAS</td>
</tr>
<tr>
<td>TMV</td>
<td>Linear Algebra</td>
<td>CUBLAS</td>
</tr>
<tr>
<td>CP</td>
<td>Scientific Computation</td>
<td>PLUTO</td>
</tr>
<tr>
<td>NBody</td>
<td>Scientific Computation</td>
<td>PLUTO</td>
</tr>
<tr>
<td>MPEG4</td>
<td>Multimedia</td>
<td>PLUTO</td>
</tr>
<tr>
<td>MRI-FH</td>
<td>Imaging</td>
<td>PLUTO</td>
</tr>
<tr>
<td>MRIQ</td>
<td>Imaging</td>
<td>PLUTO/ PARBOIL</td>
</tr>
</tbody>
</table>
## Search Space Size

<table>
<thead>
<tr>
<th>Kernel</th>
<th>I/P Comb.</th>
<th>Loop Perms. Pruned</th>
<th>O/P Comb.</th>
<th>Decom p.</th>
<th>Total Var.</th>
<th>Pruning and Autotuning Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Potential</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Variances</td>
</tr>
<tr>
<td>MM</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>MV</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>TMV</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MRIQ</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Conv</td>
<td>24</td>
<td>22</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>MRIFH</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MPEG 4</td>
<td>24</td>
<td>22</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>CP</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Nbody</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Results: SGEMM

- We compare against CUBLAS 3.2
- On GTX-280 we are within 2%
- On C2050 we are within 8%
Results: DGEMM

- We compare against CUBLAS 3.2 and MAGMA 0.2
- On GTX-280 we are within 2% of CUBLAS
- On C2050 we are within 10% of CUBLAS
Results: SGEMV

- We compare against CUBLAS 3.2
- Compiler generated code outperforms CUBLAS
- Speedup of up to 1.22x of GTX-280, 1.84x on C2050
- Raw performance of SP MV higher of GTX-280
- Similar computation partitioning and data placement strategy used
- Thread/block larger on C2050
Comparison against a state-of-the-art GPU compiler based on PLUTO

CP shows the highest speedup of 2.03x

Average speedup of 1.48X (C2050) and 1.33X (GTX-280).
Summary and Contributions

Transformation Strategy Generator (TSG) algorithm

Meta-optimizer that generates a collection of transformation recipes for parallelization targeting GPUs

Demonstrate performance portable code generation across execution contexts

High performance on two devices and across input problem sizes

Achieves high performance

Comparison with manually-tuned and state-of-the-art GPU compiler-generated code