Compiler-Based Autotuning Technology

Lecture 4: Parallel Code Generation and CUDA-CHiLL

Mary Hall
July, 2011

* This work has been partially sponsored by DOE SciDAC as part of the Performance Engineering Research Institute (PERI), DOE Office of Science, the National Science Foundation, DARPA and Intel Corporation.
Motivation

• What about parallel code?
  - Emerging multi-core and many-core architectures
  - New (CUDA and OpenCL) and old (OpenMP data-parallel programming models)
  - Heterogeneous systems

• Different CHiLL scripts can describe mapping for different architectures, applied to the same input code
  - Portable, heterogeneous support
  - Today we’ll generate CUDA code
Previous Work in Automatic Parallelization


50% higher Specfp95 ratio than previously reported

**8-processore Speedups—Digital AlphaServer 8400**

- **Old approaches:**
  - Limited to loops and array computations
  - Difficult to find sufficient granularity (parallel work between synchronization)
  - Success but from fragile, complex software

- **New ideas:**
  - Finer granularity of parallelism (more plentiful)
  - Combine with hardware support (e.g., speculation and multithreading)
  - Input from the user and autotuning
Outline for Today’s Lecture

1. Basics on GPUs and GPU code generation
2. A programming language interface adds another layer of abstraction
3. CUDA-CHiLL, Automatic Parallelization for GPUs
   a. Example, Matrix-Vector Multiply
   b. Transformation Strategy Generator
   c. Example, Matrix-Matrix Multiply for two GPU architectures
   d. Other examples: Convolution and MRI-Q
4. Productivity improvements
5. Extensive performance results
1. Nvidia GPU Basics: Two-Level Parallelism Hierarchy

Multithreaded CUDA Program
Block 0  Block 1  Block 2  Block 3  Block 4  Block 5  Block 6  Block 7

GPU with 2 Cores
Core 0  Core 1
Block 0  Block 1  Block 2  Block 3  Block 4  Block 5  Block 6  Block 7

GPU with 4 Cores
Core 0  Core 1  Core 2  Core 3
Block 0  Block 1  Block 2  Block 3  Block 4  Block 5  Block 6  Block 7

Grid
Block (0, 0)  Block (1, 0)  Block (2, 0)  Block (0, 1)  Block (1, 1)  Block (2, 1)

Block (1, 1)
Thread (0, 0)  Thread (1, 0)  Thread (2, 0)  Thread (3, 0)  Thread (0, 1)  Thread (1, 1)  Thread (2, 1)  Thread (3, 1)  Thread (0, 2)  Thread (1, 2)  Thread (2, 2)  Thread (3, 2)
1. Nvidia GPU Basics: Complex GPU Memory Hierarchy

Global Memory
- 1GB for GTX 280
- Access 100-200 cycles
- Bandwidth optimization:
  * Coalesced global memory accesses
  * Neighboring threads access adjacent data
- Also, Texture/Constant Memory

Shared Memory
- 16K per SM
- Shared across threads of SM

Register File
- 16K per SM
- Thread local data

Data Cache on Fermi only

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign

ACACES 2011, L4: Parallel code and CUDA-CHiLL
Identify two levels of parallelism

- Block-level parallelism must use very expensive synchronization to protect global memory (so should be avoided)
- Barrier synchronization available at the thread level

Data placement in the very complex memory hierarchy

- Registers for thread-local data
- "Shared" (scratchpad) memory for data shared across threads in a block
- Global memory for data with no reuse
- Texture and constant memory for read-only, reused data

Optimizations to improve memory bandwidth

- Coalesce global memory accesses
- Avoid shared memory bank conflicts
- Parallel access using texture memory
2. CUDA-CHiLL System

• Input: Sequential C Code
• Transformation Strategy Generator (TSG) automatically generates transformation recipes
  • Computational decomposition
  • Data Staging
• High-level CUDA abstraction layer generates CHiLL primitives
  • Compact scripting language
• CHiLL primitives
  • Transformation and code generation
• Autotuning and pruning of the search space

ACACES 2011, L4: Parallel code and CUDA-CHiLL
2. Higher-Level Abstraction in CUDA-CHiLL System

- High-level, programmable interface to transformation and code generation – a *programming language* approach
  - Encapsulation and control flow
  - Queries to compiler guide optimization
  - Support users with different skill levels
- **GPU Code Generation** - **CUDA-CHiLL**
  - Rapid compiler prototyping through scripting language (Lua)
  - Compact: CUDA-CHiLL is roughly 300 lines of Lua code
- **A model for other context-specific abstraction using CHiLL**
### 2. Recall CHiLL Set of Transformations

<table>
<thead>
<tr>
<th>Transformation and Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>permute ([stmt],[level],[order])</code></td>
<td>Permute optional [stmt] to optional loop [level] according to order. Can omit [stmt] and [level] and entire loop nest is permuted.</td>
</tr>
<tr>
<td><code>unroll (stmt,level,unrollfactor)</code></td>
<td>Unroll loop at level for the subloop specified by stmt/level. Unroll by unrollfactor.</td>
</tr>
<tr>
<td><code>tile (stmt,level,ts,[outerlooplevel])</code></td>
<td>Tile loop at level for the subloop specified by stmt/level and tile size ts. Place controlling loop at optional [outerlooplevel] or defaults to outermost.</td>
</tr>
<tr>
<td><code>datacopy (stmt,level,array,[index])</code></td>
<td>Calculate footprint for all references to array in subloop specified by stmt/level and copy into temporary, replacing original accesses with copy. Optional [index] refers to fastest-changing dimension.</td>
</tr>
<tr>
<td><code>split(stmt,level,condition)</code></td>
<td>Split iteration space at subloop specified by stmt/level according to condition and its complement.</td>
</tr>
<tr>
<td><code>datacopy_privatized (stmt,level,array,[index])</code></td>
<td>Similar to datacopy, but creates a private copy in parallel thread code.</td>
</tr>
<tr>
<td><strong>Other transformations include:</strong></td>
<td>fuse, distribute, skew, scale, reverse, shift, peel, nonsingular</td>
</tr>
</tbody>
</table>

ACACES 2011, L4: Parallel code and CUDA-CHiLL
## 2. CUDA-CHiLL Set of Transformations

<table>
<thead>
<tr>
<th>Command</th>
<th>Example Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tile_by_index</td>
<td>{“i”,”j”}</td>
<td>Indices of the loops to be tiled</td>
</tr>
<tr>
<td></td>
<td>{TI,TJ}</td>
<td>Tile sizes for each index variable</td>
</tr>
<tr>
<td></td>
<td>{l1_control=“ii”, l2_control= “jj”}</td>
<td>Tile controlling loop names</td>
</tr>
<tr>
<td></td>
<td>{“ii”,”jj”,”I”,”j”}</td>
<td>Final loop order</td>
</tr>
<tr>
<td>cudaize</td>
<td>“gpuMV”</td>
<td>Name of generated kernel</td>
</tr>
<tr>
<td></td>
<td>{a=N,b=N,c=N*N}</td>
<td>Sizes of input arrays</td>
</tr>
<tr>
<td></td>
<td>{block={“ii”}, thread={“jj”}}</td>
<td>Indices of blocks and threads</td>
</tr>
<tr>
<td>copy_to_registers</td>
<td>“kk”</td>
<td>Loop level for copy</td>
</tr>
<tr>
<td></td>
<td>“c”</td>
<td>Array to be copied</td>
</tr>
<tr>
<td>copy_to_shared</td>
<td>“tx”</td>
<td>Loop level for copy</td>
</tr>
<tr>
<td></td>
<td>“b”</td>
<td>Array to be copied</td>
</tr>
<tr>
<td></td>
<td>-16</td>
<td>Input to padding</td>
</tr>
<tr>
<td>Others:</td>
<td>copy_to_texture, copy_to_constant</td>
<td></td>
</tr>
<tr>
<td>unroll_to_level</td>
<td>1</td>
<td>Level of loop nest to unroll</td>
</tr>
</tbody>
</table>

ACACES 2011, L4: Parallel code and CUDA-CHiLL
2. CUDA-CHiLL Tiling Algorithm

```
TileByIndexCommands(s, I, S, M, 0)
Input:  s: Statement number; I: Indices to tile; S: Tile sizes;
       M: Map of names for indices; O: Final loop nest order
Output: F: Set of transformation operations
begin
  F := ∅
  C := extract control loop name list from M
  I' := extract renamed tile loop name map from M ∪ I
  order := BuildOrder(O, C, I', 0)
  F := F + [permute(s, order)]
for i in 1..|I| do
  level := FindLevel(I_i)
  order := BuildOrder(O, C, I', i)
  offset := offset between I'_i and C_i in order
  if offset < 0 then
    F := F + [tile(s, level, S_i, level + offset, I'_i, C_i)]
  then
    F := F + [tile(s, level, S_i, level, I'_i, C_i)]
  end
  order := BuildOrder(O, C, I', i)
  F := F + [permute(s, order)]
end
return F
```

tile_by_index(
  {"i", "j"}, {TI, TJ},
  {11_control="ii",
   12_control="jj"},
  {"ii", "jj", "i", "j"})

permute(∅, {"i", "j"})
tile(∅, 1, 64, 1, "i", "ii", 1)
permute(∅, {"ii", "i", "j"})
tile(∅, 3, 16, 2, "j", "jj", 1)
permute(∅, {"ii", "jj", "i", "j"})
3a. Example: Matrix-Vector Multiply

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i] = a[i] + c[j][i]*b[j];
3a. TSG: Computation Decomposition for Matrix-Vector Multiply

- **Block Parallelism**

  ```
  tile_by_index({"i"}, {TI}, {l1_control="ii"},{"ii","i", "j"})
  ```

  ```
  cudaize( block{"ii"}, thread{} )
  ```

- **Thread Parallelism**

  ```
  cudaize( block{"ii"}, thread{"i"} )
  ```
3a. TSG: Data Staging for Matrix-Vector Multiply

- Data Staging
  - Shared Memory
  
  ```
  tile_by_index(\{"j"\}, \{TJ\}, \{l1\_control="jj"\},\{"ii","jj","i","j"\})
  ```

- Registers

- Cudaize

- Unrolling

```latex
\text{cudaize( block{"ii"}, thread{"i"} )}
```

```latex
\text{unroll to depth( 1 )}
```

Data Reuse across threads

Data Reuse inside thread

Final Loop Order

```
// Block (N/Tl,1), Threads (Tl,1)
for (ii = 0; ii < N/Tl; ii++)    //Block.X
  for (jj = 0; jj < N/TJ; jj++)   //Tile Cont. (Shared Memory)
    for (i = 0; i < Tl; i++)     //Thread.X
      for (j = 0; j < TJ; j++)   //Thread.Y
```

\begin{equation}
a[i] = a[i] + c[j][i] \times b[j]
\end{equation}
CUDA-CHiLL Recipe

N = 1024
TI= TJ = 32
tile_by_index({"i","j"},
{TI,TJ},{l1_control="ii",
l2_control="k"},{"ii",
"jj","i", "j"})
normalize_index("i")
cudaize("mv_GPU", {a=N,
b=N, c=N*N},{block=
{"ii"}, thread={"i"}})
copy_to_shared("tx", "b", 1)
copy_to_registers("jj", "a")
unroll_to_depth(1)
3a. Matrix-Vector Multiply: GPU Code

Generated Code: with Computational decomposition only.

```c
__global__ GPU_MV(float* a, float* b, float** c) {
    int bx = blockIdx.x; int tx = threadIdx.x;
    int i = 32*bx+tx;
    for (j = 0; j < N; j++)
        a[i] = a[i] + c[j][i] * b[j];
}
```

Final MV Generated Code: with Data staged in shared memory & registers.

```c
__global__ GPU_MV(float* a, float* b, float** c) {
    int bx = blockIdx.x; int tx = threadIdx.x;
    __shared__ float bcpy[32];
    float acpy = a[tx + 32 * bx];
    for (jj = 0; jj < 32; jj++) {
        bcpy[tx] = b[32 * jj + tx];
        __syncthreads();
        //this loop is actually fully unrolled
        for (j = 32 * jj; j <= 32 * jj + 32; j++)
            acpy = acpy + c[j][32 * bx + tx] * bcpy[j];
        __syncthreads();
    }
    a[tx + 32 * bx] = acpy;
}
```

ACACES 2011, L4: Parallel code and CUDA-CHiLL
3b. TSG: 3-Phase Approach

Phase I:
Identify Candidate Computation Decompositions

Phase II:
For a given computation decomposition, identify candidates for different memory hierarchy levels

Phase III/IV: Code Gen & Autotuning
Holding computation and data partition sizes fixed (tile parameters), determine most profitable data placement for each data structure. Then, tune for tile parameters.

This approach leads to a small number of implementations, in the tens of variants.
3b. TSG: Identifying Strategies

Parallel Mapping
• Use dependence information and global memory coalescing concept to identify candidate parallel loops for blocks and threads.
• Generate tiling for block and thread decomposition.

Manage Heterogeneous Memory Hierarchy
• Register candidate: any data with reuse inside a thread.
• Shared memory candidate: any data with reuse across threads in a block AND any data with non-coalesced global memory accesses.
• Texture memory candidate: any read-only data already mapped to shared memory or registers.
• Constant memory candidate: read-only data with reuse across threads (and blocks) and short reuse distances.

Other Optimizations
• Aggressive loop unrolling by default to improve ILP, increase register reuse and reduce loop overhead.
3c. Matrix Multiply Example, Portability to Different Architectures

- Next we show the distinct implementations our autotuning framework identifies for the GTX-280 and Fermi
- Achieves comparable results to best-known previous implementations
### 3c. Comparison of Two GPUs

<table>
<thead>
<tr>
<th></th>
<th>GTX-280</th>
<th>Tesla 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>16KB</td>
<td>(up to) 48 KB</td>
</tr>
<tr>
<td>Registers/SM</td>
<td>16 K</td>
<td>32 K</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>
3c. Automatically-Generated Matrix-Matrix Multiply Scripts

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

TC2050 Fermi implementation
Mostly corresponds to CUBLAS 3.2 and MAGMA

Different computation decomposition leads to additional tile command

a in shared memory, both a and b are read through texture memory

ACACES 2011, L4: Parallel code and CUDA-CHiLL
3d. Another Example, MRI-Q

Source code from Parboil benchmark

```c
struct kValues {
  float Kx;
  float Ky;
  float Kz;
  float PhiMag;
} kVals[1];

float x[N], y[N], z[N], Qr[N], Q1[N];
for ( i = 0; i < M; i++) {
  for ( j = 0; j < N; j++) {
    expArg = P1x2 * (kVals[1].Kx*x[j] + kVals[1].Ky*y[j] + kVals[1].Kz*z[j]);
    cosArg = cosf(expArg);
    sinArg = sinf(expArg);
    phi   = kVals[1].PhiMag;
    Qr[j]  = phi * cosArg;
    Q1[j]  = phi * sinArg;
  }
}
```

Automatically-generated script

- Kernel from an MRI computation, part of Parboil suite
- Our compiler identified a different strategy that leads to higher performance than the manually-written code

ACACES 2011, L4: Parallel code and CUDA-CHiLL
3d. Another Example, 2D Convolution

Sequential Code
for(i=0;i<N;++i)
  for(j=0;j<N;++j)
    for(k=0;k<M;++k)
      for(l=0;l<M;++l)
        c[i][j] = c[i][j] + a[k+i][l+j] * b[k][l];

CUDA-CHiLL Recipe
N=4096, M=32, TI =32, TJ = 16, TI=4
permute(0,{"j","i","k","l"})
tile_by_index({"j","i"}, {TI,TJ}, {l1_control="jj",
  l2_control="ii"},{"jj", "ii", "j","i","k","l"})
normalize_index("j")
normalize_index("i")
cudaize("Kernel GPU", {a=(N+M)*(N+M), c=(N+M)*(N+M)}, {block={"j","i"}},
  {"j","i"})
copy_to_shared("tx","a",-16)
copy_to_shared("tx","b",-16)
copy_to_registers("tx", "c")
unroll_to_depth(1)

Optimized Code
__shared__ float (_P1[47])[31];
__shared__ float (_P2[16])[17]; float tmp3;
for (tmp = 16 * by + 3 * ty; tmp <= min(16 * by + 30, 16
  * by + 3 * ty + 2); tmp++)
  for (tx1 = 2 * tx; tx1 <= min(2 * tx + 1, 46); tx1++)
    _P1[ tx1][tmp - 16 * by] = a[tmp][32 * bx + tx1];
__syncthreads();
for (tmp = 0; tmp <= 15; tmp++)
  for (tx1 = 2 * tx; tx1 <= 2 * tx + 1; tx1++)
    _P2[tx1][tmp] = b[tmp][tx1];
syncthreads();
tmp3 = c[k + 16 * by][tx + 32 * bx];
for (k = 0; k <= 15; k++)
  for (l = 0; l <= 15; l++)
    tmp3 = tmp3 + _P1[l+ tx ][k + ty] * _P2[l][k];
c[k + 16 * by][tx + 32 * bx] = tmp3;

Data structures for shared memory
Complex bounds for shared memory copy loops
4. Increased Productivity of CUDA-CHiLL Interface

Simple example where tile size evenly divides problem size.

CUDA-CHiLL script (in Lua):
\[
\begin{align*}
N &= 1024 \\
TI &= 32 \\
tile\_by\_index(\{"i"\}, \{TI\}, \\
\quad \{l1\_control="ii"\},\{"ii", "i", "j"\}) \\
// normalize to start at "i"=0 \\
normalize\_index("i") \\
cudaize("mv\_GPU", \{a=N, b=N, \}
\quad c=N*N\},\{block="ii"\}, thread= \{"i"\}) \\
\text{copy\_to\_shared("tx", "b", 1)} \\
\text{copy\_to\_registers("k", "a")} \\
\text{unroll\_to\_depth(1)}
\end{align*}
\]

Equivalent CHiLL script (generated by Lua interface):
\[
\begin{align*}
NI &= 1024 \\
TI &= 32 \\
original() \\
tile(0, 1, TI, 1, i, ii, 1) \\
tile(0, 3, TI, 2, j, k, 1) \\
tile(0, 3, 3) \\
datacopy(0, 3, b, \{"tmp1","tmp2"\}, \\
\quad \text{false},0,1,1,true) \\
add\_sync(0, "i") \\
add\_sync(1, "i") \\
tile(1, 3, 3) \\
datacopy\_privatized(0, k, a, \{"i", "j"\}) \\
unroll(0, 5, 0)
\end{align*}
\]
4. Increased Productivity: Matrix Multiply

Low Level
permute(0, [i, j])
tile(0, 1, 64, 1, 1, 1, 1)
permute(0, [1, i, j])
tile(0, 4, 16, 4, t, 0, 4)
permute(0, [1, i, j, t])
cudaize("mm_GPU", [a=N*N, b=N*N, c=N*N],
{block="ii", "jj", thread="tt", "tt"})
datacopy_privateized(0, k, c, (tx, ty))
unroll(1, 5, 0)
unroll(2, 5, 0)
datacopy(0, 4, b, [tmp1, tmp2], false, 0, 1, -16, shared_mem)
add_sync(0, tx)
normalize(7, 5)
tile(7, 4, 4, tmp, ty, counted)
tile(7, 6, 14, tx, tx, counted)
add_sync(7, tx)
unroll(0, 8, 0)
unroll(3, 5, 0)
unroll(4, 5, 0)
unroll(5, 5, 0)
unroll(6, 5, 0)
unroll(7, 6, 0)
unroll(0, 8, 0)
unroll(0, 8, 0)
unroll(8, 9, 0)
unroll(8, 9, 0)
unroll(8, 9, 0)
unroll(9, 9, 0)
unroll(46, 9, 0)
unroll(59, 9, 0)
unroll(72, 9, 0)

High Level with Abstraction Layer

```
tile_by_index("i", "j", {G, T},
{11_control="ii", 12_control="jj"},
{"ii", "jj", "i", "j"})
```

```
tile_by_index("k", {T},
{11_control="kk"},
{"ii", "jj", "kk", "i", "j", "k"},
strided)
```

```
tile_by_index("i", {T},
{11_control="tt", 11_tile="t"},
{"ii", "jj", "kk", "t", "tt", "j", "k"})
```

```
cudaize("mm_GPU", [a=N*N, b=N*N, c=N*N],
{block="ii", "jj", thread="tt", "tt"})
```

```
copy_to_registers("kk", "c")
copy_to_shared("tx", "b", -16)
```

unroll_to_depth(2)

CHiLL Script Source:
Gabe Rudy
5. Performance Results: Matrix-Matrix and Matrix-Vector Multiply

(a) SGEMM GFlops
(b) DGEMM GFlops
(c) SGEMV GFlops
(d) DGEMV GFlops
5. Performance Results: Convolution and MRI-Q

2-D Convolution

MRI-Q

TC2050 Fermi

GTX-280

ACACES 2011, L4: Parallel code and CUDA-CHiLL
5. Performance Results: Impact of Different Memory Hierarchy Levels

<table>
<thead>
<tr>
<th></th>
<th>Unroll</th>
<th>Registers</th>
<th>Shared Memory</th>
<th>Texture Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>GTX280</td>
<td>Tesla C2050</td>
<td>GTX280</td>
<td>Tesla C2050</td>
</tr>
<tr>
<td>DGEMM</td>
<td>94.85</td>
<td>96.55</td>
<td>92.32</td>
<td>84.86</td>
</tr>
<tr>
<td>DGEMV</td>
<td>57.55</td>
<td>5.55</td>
<td>63.63</td>
<td>68.99</td>
</tr>
<tr>
<td>DGEMV(T)</td>
<td>64.31</td>
<td>0.63</td>
<td>20.51</td>
<td>19.94</td>
</tr>
<tr>
<td>MRIQ</td>
<td>0.00</td>
<td>1.63</td>
<td>12.47</td>
<td>11.25</td>
</tr>
<tr>
<td>Conv.</td>
<td>44.16</td>
<td>47.46</td>
<td>77.29</td>
<td>54.29</td>
</tr>
</tbody>
</table>

- Table shows reduction in performance when specific level of the memory hierarchy is not used.
5. Performance Results: A Few More Details

- Generated code is very long, over 1000 lines of code for some versions of Matrix-Matrix Multiply
  - Problem size may not be evenly divided by computation decomposition parameters
  - Cleanup code for tiling and unrolling can be lengthy, important to optimize too

- No more than 32 versions needed to generalize
Summary of Lecture

• Two ideas
  - A programming language interface to CHiLL’s primitives allows custom higher-level abstractions
  - Used to develop CUDA-CHiLL abstractions for auto-tuning high-performance GPU code

• Features
  - TSG generates multiple possible implementations that are searched using autotuning
  - Performance: can sometimes outperform CUBLAS and manual code
References

Other GPU compiler frameworks.

CUDA-CHiLL reference.

ACACES 2011, L4: Parallel code and CUDA-CHiLL