CUDA Application Development

Wen-mei Hwu
University of Illinois, Urbana-Champaign
# Science and Engineering Application Speedup

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>H.264</td>
<td>Registers, global memory latency</td>
<td>3,936</td>
<td>20.2</td>
<td>1.5</td>
</tr>
<tr>
<td>LBM</td>
<td>Shared memory capacity</td>
<td>3,200</td>
<td>12.5</td>
<td>12.3</td>
</tr>
<tr>
<td>RC5-72</td>
<td>Registers</td>
<td>3,072</td>
<td>17.1</td>
<td>11.0</td>
</tr>
<tr>
<td>FEM</td>
<td>Global memory bandwidth</td>
<td>4,096</td>
<td>11.0</td>
<td>10.1</td>
</tr>
<tr>
<td>RPES</td>
<td>Instruction issue rate</td>
<td>4,096</td>
<td>210.0</td>
<td>79.4</td>
</tr>
<tr>
<td>PNS</td>
<td>Global memory capacity</td>
<td>2,048</td>
<td>24.0</td>
<td>23.7</td>
</tr>
<tr>
<td>LINPACK</td>
<td>Global memory bandwidth, CPU-GPU data transfer</td>
<td>12,288</td>
<td>19.4</td>
<td>11.8</td>
</tr>
<tr>
<td>TRACF</td>
<td>Shared memory capacity</td>
<td>4,096</td>
<td>60.2</td>
<td>21.6</td>
</tr>
<tr>
<td>FDTD</td>
<td>Global memory bandwidth</td>
<td>1,365</td>
<td>10.5</td>
<td>1.2</td>
</tr>
<tr>
<td>MRI-FHD</td>
<td>Instruction issue rate</td>
<td>8,192</td>
<td>23.0</td>
<td>23.0</td>
</tr>
</tbody>
</table>

[HKR HotChips-2007]
Chemo Therapy Monitoring

6-12 weeks
**MRI Reconstruction**

Spiral scan data + Iterative recon: Fast scan reduces artifacts, iterative reconstruction increases SNR. Reconstruction requires a lot of computation.
An Exciting Revolution - Sodium Map of the Brain

- Images of sodium in the brain
  - Requires powerful scanner (9.4 Tesla)
  - Very large number of samples for increased SNR
  - Requires high-quality reconstruction

- Enables study of brain-cell viability before anatomic changes occur in stroke and cancer treatment - within days!

Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago
Advanced MRI Reconstruction

\[(F^H F + \lambda W^H W) \rho = F^H d\]

- Q depends only on scanner configuration
- \(F^H d\) depends on scan data
- \(\rho\) found using linear solver
  - \(F^H F\) computed once per iteration; depends on Q, \(F^H d\)
  - \(\lambda W^H W\) incorporates anatomical constraints

Reconstruction of a 64\(^3\) image used to take days!

Final Data Arrangement and Fast Math

Performance: 128 GFLOPS
Time: 1.2 minutes

```
exp = x[p] * s[d].kx + y[p] * s[d].ky + z[p] * s[d].kz;
cArg = cos(exp);
sArg = sin(exp);
rFhD[p] += cArg * s[d].rRho - sArg * s[d].iRho;
iFhD[p] += cArg * s[d].iRho + sArg * s[d].rRho;
```
for (m = 0; m < M; m++) {
    phi[m] = rPhi[m]*rPhi[m] + iPhi[m]*iPhi[m]

    for (n = 0; n < N; n++) {
        exp = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n])
        rQ[n] += phi[m]*cos(exp)
        iQ[n] += phi[m]*sin(exp)
    }
}

• $F^h_d$ is nearly identical
• Scan data
  • $M = \#$ scan points
  • $k_x, k_y, k_z = 3D$ scan data
• Pixel data
  • $N = \#$ pixels
  • $x, y, z = \text{input 3D pixel data}$
  • $Q = \text{output pixel data}$
• Complexity is $O(MN)$
• Inner loop
  • 10 FP MUL or ADD ops
  • 2 FP trig ops
  • 10 loads
for (m = 0; m < M; m++) {
    phi[m] = rPhi[m]*rPhi[m] +
             iPhi[m]*iPhi[m]

    for (n = 0; n < N; n++) {
        exp = 2*PI*(kx[m]*x[n] +
                     ky[m]*y[n] +
                     kz[m]*z[n])
        rQ[n] += phi[m]*cos(exp)
        iQ[n] += phi[m]*sin(exp)
    }
}
for (m = 0; m < M; m++) {
    phi[m] = rPhi[m]*rPhi[m] +
            iPhi[m]*iPhi[m]
    for (n = 0; n < N; n++) {
        exp = 2*PI*(kx[m]*x[n] +
                     ky[m]*y[n] +
                     kz[m]*z[n])
        rQ[n] += phi[m]*cos(exp)
        iQ[n] += phi[m]*sin(exp)
    }
}

for (n = 0; n < N; n++) {
    for (m = 0; m < M; m++) {
        phi[m] = rPhi[m]*rPhi[m] +
                 iPhi[m]*iPhi[m]
        exp = 2*PI*(kx[m]*x[n] +
                     ky[m]*y[n] +
                     kz[m]*z[n])
        rQ[n] += phi[m]*cos(exp)
        iQ[n] += phi[m]*sin(exp)
    }
}
From C to CUDA: Step 1
What unit of work is assigned to each thread?

for (n = 0; n < N; n++) {
    for (m = 0; m < M; m++) {
        phi[m] = rPhi[m]*rPhi[m]
            + iPhi[m]*iPhi[m]
        exp = 2*PI*(kx[m]*x[n] +
            ky[m]*y[n] +
            kz[m]*z[n])
        rQ[n] += phi[m]*cos(exp)
            iQ[n] += phi[m]*sin(exp)
    }
}

for (m = 0; m < M; m++) {
    phi[m] = rPhi[m]*rPhi[m]
        + iPhi[m]*iPhi[m]
}

for (n = 0; n < N; n++) {
    for (m = 0; m < M; m++) {
        exp = 2*PI*(kx[m]*x[n] +
            ky[m]*y[n] +
            kz[m]*z[n])
        rQ[n] += phi[m]*cos(exp)
            iQ[n] += phi[m]*sin(exp)
    }
}

How does loop fission help?
From C to CUDA: Step 1
What unit of work is assigned to each thread?

for (m = 0; m < M; m++) {
    phi[m] = rPhi[m]*rPhi[m] + iPhi[m]*iPhi[m]
}

for (n = 0; n < N; n++) {
    for (m = 0; m < M; m++) {
        exp = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n])
        rQ[n] += phi[m]*cos(exp)
        iQ[n] += phi[m]*sin(exp)
    }
}

- phi kernel
  Each thread computes phi at one scan point (each thread corresponds to one loop iteration)

- Q kernel
  Each thread computes Q at one pixel (each thread corresponds to one outer loop iteration)
for (m = 0; m < M/32; m++) {
    exp = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
}
for (m = 31M/32; m < 32M/32; m++)
{
    exp = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
}
From C to CUDA: Step 2
Where are the potential bottlenecks?

```c
Q(float* x, y, z, rQ, iQ, kx, ky, kz, phi, int startM, endM)
{
    n = blockIdx.x*TPB + threadIdx.x
    for (m = startM; m < endM; m++) {
        exp = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n])
        rQ[n] += phi[m] * cos(exp)
        iQ[n] += phi[m] * sin(exp)
    }
}
```

Bottlenecks
- Memory BW
- Trig ops
- Overheads (branches, addr calcs)
Step 3: Overcoming bottlenecks

- LS recon on CPU (SP)
- Q: 45 hours, 0.5 GFLOPS
- $F^H_d$: 7 hours, 0.7 GFLOPS
- Counting each trig op as 1 FLOP
Step 3: Overcoming Bottlenecks (Mem BW)

- Register allocate pixel data
  - Inputs (x, y, z); Outputs (rQ, iQ)
- Exploit temporal and spatial locality in access to scan data
  - Constant memory + constant caches
  - Shared memory
Step 3: Overcoming Bottlenecks (Mem BW)

- Register allocation of pixel data
  - Inputs (x, y, z); Outputs (rQ, iQ)
  - FP arithmetic to off-chip loads: 2 to 1
- Performance
  - 5.1 GFLOPS (Q), 5.4 GFLOPS (F\text{Hd})
  - Still bottlenecked on memory BW
Step 3: Overcoming Bottlenecks (Mem BW)

- **Old bottleneck: off-chip BW**
  - Solution: constant memory
  - FP arithmetic to off-chip loads: 284 to 1

- **Performance**
  - 18.6 GFLOPS (Q), 22.8 GFLOPS (F^Hd)

- **New bottleneck: trig operations**
Sidebar: Estimating Off-Chip Loads with Const Cache

- How can we approximate the number of off-chip loads when using the constant caches?
- Given: 128 tpb, 4 blocks per SM, 256 scan points per grid
- Assume no evictions due to cache conflicts
- 7 accesses to global memory per thread (x, y, z, rQ x 2, iQ x 2)
  - 4 blocks/SM * 128 threads/block * 7 accesses/thread = 3,584 global mem accesses
- 4 accesses to constant memory per scan point (kx, ky, kz, phi)
  - 256 scan points * 4 loads/point = 1,024 constant mem accesses
- Total off-chip memory accesses = 3,584 + 1,024 = 4,608
- Total FP arithmetic ops = 4 blocks/SM * 128 threads/block * 256 iters /thread * 10 ops/iter = 1,310,720
- FP arithmetic to off-chip loads: 284 to 1
Step 3: Overcoming Bottlenecks (Trig)

- Old bottleneck: trig operations
  - Solution: SFUs
- Performance
  - 98.2 GFLOPS (Q), 92.2 GFLOPS ($F^Hp$)
- New bottleneck: overhead of branches and address calculations
Sidebar: Effects of Approximations

- Avoid temptation to measure only absolute error ($I_0 - I$)
  - Can be deceptively large or small
- Metrics
  - PSNR: Peak signal-to-noise ratio
  - SNR: Signal-to-noise ratio
- Avoid temptation to consider only the error in the computed value
  - Some apps are resistant to approximations; others are very sensitive

$$MSE = \frac{1}{mn} \sum_i \sum_j (I(i, j) - I_0(i, j))^2$$

$$A_s = \frac{1}{mn} \sum_i \sum_j I_0(i, j)^2$$

$$PSNR = 20 \log_{10} \left( \frac{\text{max}(I_0(i, j))}{\sqrt{MSE}} \right)$$

$$SNR = 20 \log_{10} \left( \frac{\sqrt{A_s}}{\sqrt{MSE}} \right)$$

Step 3: Overcoming Bottlenecks (Overheads)

- Old bottleneck: Overhead of branches and address calculations
  - Solution: Loop unrolling and experimental tuning
- Performance
  - 179 GFLOPS (Q), 145 GFLOPS (F^H d)
Experimental Tuning: Tradeoffs

• In the Q kernel, three parameters are natural candidates for experimental tuning
  • Loop unrolling factor (1, 2, 4, 8, 16)
  • Number of threads per block (32, 64, 128, 256, 512)
  • Number of scan points per grid (32, 64, 128, 256, 512, 1024, 2048)

• Can’t optimize these parameters independently
  • Resource sharing among threads (register file, shared memory)
  • Optimizations that increase a thread’s performance often increase the thread’s resource consumption, reducing the total number of threads that execute in parallel

• Optimization space is not linear
  • Threads are assigned to SMs in large thread blocks
  • Causes discontinuity and non-linearity in the optimization space
Increase in per-thread performance, but fewer threads:
Lower overall performance
Experimental Tuning: Scan Points Per Grid

![Graph showing time (s) vs. scan points per grid](image)
Sidebar: Cache-Conscious Data Layout

Scan Data |
---|
\[ kx[i] \] \( kx \) |
\[ ky[i] \] \( ky \) |
\[ k\phi[i] \] \( kz \) |
\[ \phi[i] \] \( \phi \) 

Scan Data |
---|
\[ kx[i] \] \( kx[i] \) |
\[ ky[i] \] \( ky[i] \) |
\[ k\phi[i] \] \( kz[i] \) |
\[ \phi[i] \] \( \phi[i] \) 

Constant Memory

- \( kx, ky, kz, \) and \( \phi \) components of same scan point have spatial and temporal locality
  - Prefetching
  - Caching
- Old layout does not fully leverage that locality
- New layout does fully leverage that locality
Experimental Tuning:
Scan Points Per Grid (Improved Data Layout)
Experimental Tuning: Loop Unrolling Factor
Sidebar: Optimizing the CPU Implementation

- Optimizing the CPU implementation of your application is very important
  - Often, the transformations that increase performance on CPU also increase performance on GPU (and vice-versa)
  - The research community won’t take your results seriously if your baseline is crippled

- Useful optimizations
  - Data tiling
  - SIMD vectorization (SSE)
  - Fast math libraries (AMD, Intel)
  - Classical optimizations (loop unrolling, etc)

- Intel compiler (icc, icpc)
## Summary of Results

<table>
<thead>
<tr>
<th>Reconstruction</th>
<th>Q Run Time (m)</th>
<th>GFLOP</th>
<th>F^{Hd} Run Time (m)</th>
<th>GFLOP</th>
<th>Linear Solver (m)</th>
<th>Recon. Time (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gridding + FFT (CPU, DP)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.39</td>
</tr>
<tr>
<td>LS (CPU, DP)</td>
<td>4009.0</td>
<td>0.3</td>
<td>518.0</td>
<td>0.4</td>
<td>1.59</td>
<td>519.59</td>
</tr>
<tr>
<td>LS (CPU, SP)</td>
<td>2678.7</td>
<td>0.5</td>
<td>342.3</td>
<td>0.7</td>
<td>1.61</td>
<td>343.91</td>
</tr>
<tr>
<td>LS (GPU, Naïve)</td>
<td>260.2</td>
<td>5.1</td>
<td>41.0</td>
<td>5.4</td>
<td>1.65</td>
<td>42.65</td>
</tr>
<tr>
<td>LS (GPU, CMem)</td>
<td>72.0</td>
<td>18.6</td>
<td>9.8</td>
<td>22.8</td>
<td>1.57</td>
<td>11.37</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU)</td>
<td>13.6</td>
<td>98.2</td>
<td>2.4</td>
<td>92.2</td>
<td>1.60</td>
<td>4.00</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU, Exp)</td>
<td>7.5</td>
<td>178.9</td>
<td>1.5</td>
<td>144.5</td>
<td>1.69</td>
<td>3.19</td>
</tr>
</tbody>
</table>

The table shows the run time, GFLOP, and linear solver time for different reconstruction methods. The Recon. Time (m) column highlights the most efficient method, which is LS (GPU, CMem, SFU, Exp) with a time of 3.19 seconds. This is 8X faster than the Gridding + FFT (CPU, DP) method.
## Summary of Results

<table>
<thead>
<tr>
<th>Reconstruction</th>
<th>Q</th>
<th>F(^H)d</th>
<th>Linear Solver (m)</th>
<th>Recons. Time (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run Time (m)</td>
<td>GFLOP</td>
<td>Run Time (m)</td>
<td>GFLOP</td>
</tr>
<tr>
<td>Gridding + FFT (CPU, DP)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LS (CPU, DP)</td>
<td>4009.0</td>
<td>0.3</td>
<td>518.0</td>
<td>0.4</td>
</tr>
<tr>
<td>LS (CPU, SP)</td>
<td>2678.7</td>
<td>0.5</td>
<td>342.3</td>
<td>0.7</td>
</tr>
<tr>
<td>LS (GPU, Naïve)</td>
<td>260.2</td>
<td>5.1</td>
<td>41.0</td>
<td>5.4</td>
</tr>
<tr>
<td>LS (GPU, CMem)</td>
<td>72.0</td>
<td>18.6</td>
<td>9.8</td>
<td>22.8</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU)</td>
<td>13.6</td>
<td>98.2</td>
<td>2.4</td>
<td>92.2</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU, Exp)</td>
<td>7.5</td>
<td>178.9</td>
<td>1.5</td>
<td>144.5</td>
</tr>
</tbody>
</table>

**Speedup Comparison:**
- **357X**
- **228X**
- **108X**
Results must be validated by domain experts.
Now, the coming battles.

“There is always hope.”
- In the eve of the Battle of the Helms Deep
Battle #1 - Tools need to do more heavy lifting!

• Programmers are doing too much heavy lifting

• Too many memory organizational details are exposed to the programmers

• Search spaces can be huge! Exhaustive search with smaller-than-usual data sets still takes significant time.

By selecting only Pareto-optimal points, we pruned the search space by 98% and still found the optimal configuration.

Implicitly parallel programming with data structure and function property annotations to enable auto parallelization

Locality annotation programming to eliminate need for explicit management of memory types and data transfers

Parameterized CUDA programming using auto-tuning and optimization space pruning

1st generation CUDA programming with explicit, hardwired thread organizations and explicit management of memory types and data transfers
Battle #2 – Reaching deeper into apps.

- Many data parallel apps have a small number of simple, dominating components
  - Low hanging fruit for parallel computing (meat)

- Small computation components often dominate after the low hanging fruits are picked
  - Usually much more difficult to parallelize (pit)
Performance of Advanced MRI Reconstruction

10 kernels, less than 1.5 min after acceleration

What does it take to reach deeper?

- MRI: Launching multiple kernels
  - 3D FFT formulated as multiple 2D FFTs.
  - Multiple kernel types beneficial in some apps

- Global Synchronization
  - Some apps require global synch at time step boundaries

- Atomic memory operations
  - Shared memory, device global memory

- Less tedious tuning process for kernels
  - Developers run out of time!
Battle #3: There is room for only one model.

- App developers cannot afford to write their apps in multiple programming models
  - It is all about programmer acceptance!

- Code based on the winning model will need to work well on both GPUs and Multi-cores
  - CUDA kernels currently only work on NVIDIA GPUs
  - How about ATI GPUs, Intel Multi-cores, and AMD Multi-cores?
**MCUDA: Thread Blocks to CPU Threads**

- A single GPU thread is too small for a CPU Thread
  - CUDA emulation does this and performs poorly

- CPU cores designed for ILP, SIMD
  - Optimizing compilers work well with iterative loops

- Turn GPU thread blocks from CUDA into iterative CPU loops
Key Issue: Synchronization

- Suspend and Wakeup
- Move on to other threads
- Begin again after all hit barrier

```c
Matrixmul(A[ ], B[ ], C[ ])
{
    __shared__ Asub[ ][ ], Bsub[ ][ ];
    int a,b,c;
    float Csub;
    int k;
    ...
    for(…)
    {
        Asub[tx][ty] = A[a];
        Bsub[tx][ty] = B[b];
        __syncthreads();
        for( k = 0; k < blockDim.x; k++ )
            Csub += Asub[ty][k] + Bsub[k][tx];
        __syncthreads();
    }
    ...
}
```
Synchronization solution

- Loop fission
  - Break the loop into multiple smaller loops according to syncthreads()
  - Not always possible with syncthreads() in control flow
**Bigger Picture Performance Results**
(reduced Sample volume)

- Consistent speed-up over hand-tuned single-thread code
- Best optimizations for GPU and CPU not always the same

<table>
<thead>
<tr>
<th>Application</th>
<th>C on CPU Time</th>
<th>CUDA on CPU Time</th>
<th>Speedup*</th>
<th>CUDA on G80 Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI-FHD</td>
<td>~1000s</td>
<td>230s</td>
<td>~4x</td>
<td>8.5s</td>
</tr>
<tr>
<td>CP</td>
<td>180s</td>
<td>45s</td>
<td>4x</td>
<td>.28s</td>
</tr>
<tr>
<td>SAD</td>
<td>42.5ms</td>
<td>25.6ms</td>
<td>1.66x</td>
<td>4.75ms</td>
</tr>
<tr>
<td>MM (4Kx4K)</td>
<td>7.84s**</td>
<td>10.1s</td>
<td>3.69x</td>
<td>1.12s</td>
</tr>
</tbody>
</table>

*Over hand-optimized CPU
**Intel MKL, multi-core execution
To Learn More

- UIUC ECE498AL - Programming Massively Parallel Processors
  (http://courses.ece.uiuc.edu/ece498/al/)
  - David Kirk (NVIDIA) and Wen-mei Hwu (UIUC) co-instructors
  - CUDA programming, GPU computing, lab exercises, and projects
  - Lecture slides and voice recordings
- More than 500 students worldwide follow the course each semester.
Thank you! Any questions?