The Art of GPU Performance
Threading & Memory Hierarchy

David Porter
“I feel the need … for speed!”

• Thread & Memory Hierarchies

• Test problem: N-body force calculation
  – 3 versions
  – Uses of memory hierarchy
  – Performance vs. threading and problem size

• Asynchronous Device (if there is time)
CUDA Thread Hierarchy

- **Grid**: Invoked by a call to device Kernel code
  - `mysub<<<BPG, TPB>>>(...);`
  - Generates `PBG*TPB` instances of the `mysub` routine

- **Block**: `BPG= Number of “Blocks Per Grid”`
  - `TPB` threads run concurrently in a block

- **Thread**: `TPB = Number of “Threads Per Block”`
  - Each thread is one instance of the `mysub` routine

Example:
- `BPG=4`
- `TPB=5`
CUDA Memory Hierarchy on GPU

- Per-thread local memory
  - Private to each thread

- Per block shared memory
  - Shared between threads in a block
  - Private to each block

- Global memory (on device)
  - Shared between threads
  - Shared between blocks
  - Shared between grids
  - Lasts till device is reset
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprocessors</td>
<td>15</td>
</tr>
<tr>
<td>CUDA CORES/MP</td>
<td>32</td>
</tr>
<tr>
<td>Total CUDA Cores</td>
<td>480</td>
</tr>
<tr>
<td>Max threads per block</td>
<td>1024</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>GPU Clock Speed</td>
<td>1.40 GHz</td>
</tr>
<tr>
<td>Memory Clock rate</td>
<td>1848.00 Mhz</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>384-bit</td>
</tr>
<tr>
<td>Total global memory:</td>
<td>1536 MBytes</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>786432 bytes</td>
</tr>
<tr>
<td>Constant memory</td>
<td>65536 bytes</td>
</tr>
<tr>
<td>Shared memory per block</td>
<td>49152 bytes</td>
</tr>
<tr>
<td>Registers per block</td>
<td>32768</td>
</tr>
</tbody>
</table>

Threads scheduled 32 at a time in “warps”

Max threads per block 1024

Shared mem per block 48 KB
Amazing performance

- NVIDIA SDK “nbody” example
- In NVIDIA_GPU_Computing_SDK
  Source: ./C/src/nbody
  Run: ./C/bin/linux/releases/nbody
N-Body Force Calculation

\[ F_i = GM_i \sum_{j=1}^{N} \frac{M_j (\vec{x}_j - \vec{x}_i)}{(d_s^2 + |\vec{x}_j - \vec{x}_i|^2)^{3/2}} \]

- For each of N bodies:
  - sum forces from all other bodies
- Every body interacts with all others
  - Total work scales as: N*N
  - Intensive access of memory across all N bodies
#include <stdio.h>
#include <cutil_inline.h>

// struct float3 { float x, y, z; }

// host Variables
float3* h_pos;
float3* h_frc;

// device Variables
float3* d_pos;
float3* d_frc;

Struct float3 = a 3D vector

Pointers for host arrays
Position: h_pos
Forrces: h_frc

Pointers for device arrays
Mirror of host arrays
int main(int argc, char** argv)
{
    // Run parameters from command line
    int N=1, threadsPerBlock = 128;
    if(argc > 1) sscanf(argv[1], "%d", &N);
    if(argc > 2) sscanf(argv[2], "%d", &threadsPerBlock);

    // Initialize events for timing GPU
    cudaEvent_t start, stop;
    cudaEventCreate( &start );
    cudaEventCreate( &stop );

    Host code
    Entry point for app

    N=Number of bodies
    ThreadsPerBlock

    Timer events
    Declared on device
    with pointer to
    reference on host
Allocate & Fill Arrays

// Host
size_t size3 = 3 * N * sizeof(float);
h_pos = (float3*)malloc(size3);
RandomInitSphere(h_pos, N, 100.0);

// Device
cudaMalloc((void**)&d_pos, size3);
cudaMalloc((void**)&d_frc, size3);
cudaMemcpy(d_pos, h_pos, size3, cudaMemcpyHostToDevice);

data

Position and force arrays: each is an array 3-vector

For single precision, memory of each array is: 12*N bytes
void RandomInitSphere(float3* data, int n, float radius) {
    for (int i = 0; i < n; i++) {
        float x=2, y=2, z=2;
        while(x*x + y*y + z*z > 1.0) {
            x = 2.0 * (rand() / (float)RAND_MAX - 0.5);
            y = 2.0 * (rand() / (float)RAND_MAX - 0.5);
            z = 2.0 * (rand() / (float)RAND_MAX - 0.5);
        }
        data[i].x = radius * x;
        data[i].y = radius * y;
        data[i].z = radius * z;
    }
}

Positions randomly sample a sphere
Run Test with Timing

// Invoke Kernel
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
cudaEventRecord( start, 0 );
CalcForces0<<<blocksPerGrid, threadsPerBlock>>>(d_pos, d_frc, N);
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

// Retrieve internal GPU timing
float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop );

CalcForce0 is kernel code: runs on the device
Device events “start” and “stop” used to time force calculation
Host & device must be synchronized at stop event
Otherwise host might retrieve value of stop BEFORE it is set
Calculate & Report Performance

```c
float pairs = (float)N * (float)N;
float flops = (float)flop_per_pair * pairs;
float timesec = elapsedTime / (1000.0);
float gflops = (flops/1000000000.0)/timesec;

printf("%9d %9d %8d %10.3f %10.3f\n",
      flop_per_pair, N, threadsPerBlock, timesec, gflops);
```

Simple CalcForce0 code calculates N-terms per body
Includes self term – no singularity because of force softening

We will see that flop_per_pair = 20
if (h_pos) free(h_pos);      // Free host memory
if (d_pos) cudaFree(d_pos);  // Free device memory
if (d_frc) cudaFree(d_frc);
cutilDeviceReset();          // Reset GPU

    return 0;
}                           // End of main routine
Kernel Code CalcForce0: Setup

```c
__global__ void CalcForces0(const float* pos, float* force, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    float x, y, z, dinv, ffac;
    float gravmass = 2.12;  // G*mass
    float ds     = 1.23;   // softening length

    if (i < N) {
        /*** WORK GOES HERE   ***/
    }
}
```

Values in blockDim.x, blockIdx.x, threadIdx.x provided
Value of i is unique on each thread and runs from 0 to N-1
Kernel Code CalcForces0: Work

```c
force[i].x = 0.0;
force[i].y = 0.0;
force[i].z = 0.0;
for(int j=0; j<N; j++) {
    x = pos[j].x - pos[i].x;
    y = pos[j].y - pos[i].y;
    z = pos[j].z - pos[i].z;
    dinv = rsqrtf(ds + x*x + y*y + z*z);
    ffac = gravmass * dinv * dinv * dinv;
    force[i].x += x*ffac;
    force[i].y += y*ffac;
    force[i].z += z*ffac;
}
```

Floating point ops per force pair

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ or -</td>
<td>9</td>
</tr>
<tr>
<td>* or /</td>
<td>9</td>
</tr>
<tr>
<td>rsqrtf</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>20</td>
</tr>
</tbody>
</table>

#define flop_per_pair 20

Reads/Writes to force[i] → Traffic to global mem
### CalcForces0 Performance vs. Size: T/B=320

<table>
<thead>
<tr>
<th># Flop/pair</th>
<th>N</th>
<th>#threads</th>
<th>Time[sec]</th>
<th>GFlop/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1024</td>
<td>320</td>
<td>0.001</td>
<td>30.775</td>
</tr>
<tr>
<td>20</td>
<td>2048</td>
<td>320</td>
<td>0.001</td>
<td>59.124</td>
</tr>
<tr>
<td>20</td>
<td>4096</td>
<td>320</td>
<td>0.003</td>
<td>99.272</td>
</tr>
<tr>
<td>20</td>
<td>8192</td>
<td>320</td>
<td>0.013</td>
<td>102.429</td>
</tr>
<tr>
<td>20</td>
<td>16384</td>
<td>320</td>
<td>0.052</td>
<td>103.240</td>
</tr>
<tr>
<td>20</td>
<td>32768</td>
<td>320</td>
<td>0.189</td>
<td>113.686</td>
</tr>
<tr>
<td>20</td>
<td>65536</td>
<td>320</td>
<td>0.747</td>
<td>114.988</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>320</td>
<td>2.936</td>
<td>117.042</td>
</tr>
</tbody>
</table>

- Treads per block (T/B) = 320 – near optimal value
- Force calculation on N=131,072 bodies takes nearly 3 sec.
- Performance << 1400GFlop/sec theoretical for the GTX 480
float fx = 0.0, fy = 0.0, fz = 0.0;
for(int j=0; j<N; j++) {
    x = pos[j].x - pos[i].x;
    y = pos[j].y - pos[i].y;
    z = pos[j].z - pos[i].z;
    dinv = rsqrtf(ds + x*x + y*y + z*z);
    ffac = gravmass * dinv * dinv * dinv;
    fx += x * ffac;
    fy += y * ffac;
    fz += z * ffac;
}
force[i].x = fx; force[i].y = fy; force[i].z = fz;

Most Reads/Writes now go to thread local fx, fy, fz
Instead of d_frc global array

Much less Traffic to global shared mem
### CalcForces1 Performance vs. Size: T/B=320

<table>
<thead>
<tr>
<th># Flop/pair</th>
<th>N</th>
<th>#threads</th>
<th>Time[sec]</th>
<th>GFlop/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1024</td>
<td>320</td>
<td>0.000</td>
<td>69.593</td>
</tr>
<tr>
<td>20</td>
<td>2048</td>
<td>320</td>
<td>0.001</td>
<td>147.206</td>
</tr>
<tr>
<td>20</td>
<td>4096</td>
<td>320</td>
<td>0.001</td>
<td>303.188</td>
</tr>
<tr>
<td>20</td>
<td>8192</td>
<td>320</td>
<td>0.003</td>
<td>397.402</td>
</tr>
<tr>
<td>20</td>
<td>16384</td>
<td>320</td>
<td>0.013</td>
<td>418.197</td>
</tr>
<tr>
<td>20</td>
<td>32768</td>
<td>320</td>
<td>0.051</td>
<td>420.534</td>
</tr>
<tr>
<td>20</td>
<td>65536</td>
<td>320</td>
<td>0.203</td>
<td>422.541</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>320</td>
<td>0.713</td>
<td>482.031</td>
</tr>
<tr>
<td>20</td>
<td>262144</td>
<td>320</td>
<td>2.836</td>
<td>484.607</td>
</tr>
</tbody>
</table>

Factor of ~4 speedup over CalcForce0 code

Performance now dramatically increases with problem size
### CalcForces1 Performance vs. T/B: N=131072

<table>
<thead>
<tr>
<th># Flop/pair</th>
<th>N</th>
<th>#threads</th>
<th>Time[sec]</th>
<th>GFlop/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>131072</td>
<td>32</td>
<td>1.224</td>
<td>280.655</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>64</td>
<td>0.739</td>
<td>465.035</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>96</td>
<td>0.709</td>
<td>484.376</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>128</td>
<td>0.704</td>
<td>487.796</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>160</td>
<td>0.709</td>
<td>484.473</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>192</td>
<td>0.730</td>
<td>470.603</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>224</td>
<td>0.744</td>
<td>461.915</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>256</td>
<td>0.728</td>
<td>471.852</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>288</td>
<td>0.761</td>
<td>451.436</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>320</td>
<td>0.713</td>
<td>482.031</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>352</td>
<td>0.778</td>
<td>441.457</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>384</td>
<td>0.730</td>
<td>470.571</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>416</td>
<td>0.721</td>
<td>476.774</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>448</td>
<td>0.746</td>
<td>460.865</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>480</td>
<td>0.770</td>
<td>446.116</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>512</td>
<td>0.731</td>
<td>470.222</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>768</td>
<td>0.732</td>
<td>469.707</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>1024</td>
<td>0.733</td>
<td>468.437</td>
</tr>
</tbody>
</table>

For large N, performance is weakly dependent on threadsPerBlock when T/B > 32.
CalcForces1 Performance vs. T/B & N

- Increase with size (N)
- Decreases with T/B
- Best speeds: Gflop/s ~ 468
  - T/B < 400
  - N > 65,000
CalcForces2: Routine get_pos

```c
__device__ void get_pos(float3 *sub,
       const float3 *full, int j0, int j, int Ns)
{
    if(j < Ns) {
        sub[j].x = full[j0+j].x;
        sub[j].y = full[j0+j].y;
        sub[j].z = full[j0+j].z;
    }
}
```

**Pure device code**

**Just copies a 3-vector FROM global h_pos TO block local "sub"**

**If test guards against idle threads**

**In case threadsPerBlock does not divide evenly into N**

© 2009 Regents of the University of Minnesota. All rights reserved.
CalcForces2: Thread Evaluates force[i]

float fx = 0.0;  fy = 0.0,  fz = 0.0;
float x0 = pos[i].x,  y0 = pos[i].y, z0 = pos[i].z;

for(int j0=0; j0<N; j0 += blockDim.x) {
    int j1 = j0 + blockDim.x; if(j1 > N) j1 = N;
    int Ns = j1 - j0;

    __shared__ float3 pos_sub[1024];
    get_pos(pos_sub, pos, j0, threadIdx.x, Ns);
    __syncthreads();

    /*** Inner Work Loop: sums to fx, fy, fz ***/
}
force[i].x = fx; force[i].y = fy; force[i].z = fz;

Thread local variables

Loop over sets of bodies
[j0, j0+blockDim.x-1]

Block shared: pos_sub
Copy h_pos once per block

Force sums done here

Thread local fx, fy, fz copied to global force array
CalcForces2: Inner Work Loop

for(int j=0; j<Ns; j++) {
    x = pos_sub[j].x - x0;
    y = pos_sub[j].y - y0;
    z = pos_sub[j].z - z0;
    dinv = rsqrtf(ds + x*x + y*y + z*z);
    ffac = gravmass * dinv * dinv * dinv;
    fx += x * ffac;
    fy += y * ffac;
    fz += z * ffac;
}
__syncthreads();

Same work per pair as before

Need to synchronize threads
All threads must be done with current pos_sub before it is refilled with positions of the next set of Ns bodies.
CalcForces2 Performance vs. Size: T/B=320

<table>
<thead>
<tr>
<th># Flop/pair</th>
<th>N</th>
<th>#threads</th>
<th>Time[sec]</th>
<th>GFlop/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1024</td>
<td>320</td>
<td>0.000</td>
<td>74.524</td>
</tr>
<tr>
<td>20</td>
<td>2048</td>
<td>320</td>
<td>0.001</td>
<td>159.552</td>
</tr>
<tr>
<td>20</td>
<td>4096</td>
<td>320</td>
<td>0.001</td>
<td>329.886</td>
</tr>
<tr>
<td>20</td>
<td>8192</td>
<td>320</td>
<td>0.003</td>
<td>434.756</td>
</tr>
<tr>
<td>20</td>
<td>16384</td>
<td>320</td>
<td>0.012</td>
<td>452.162</td>
</tr>
<tr>
<td>20</td>
<td>32768</td>
<td>320</td>
<td>0.042</td>
<td>517.217</td>
</tr>
<tr>
<td>20</td>
<td>65536</td>
<td>320</td>
<td>0.166</td>
<td>516.170</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>320</td>
<td>0.663</td>
<td>518.357</td>
</tr>
<tr>
<td>20</td>
<td>262144</td>
<td>320</td>
<td>2.603</td>
<td>527.954</td>
</tr>
</tbody>
</table>

Over 500 GFlop/sec on large problem sizes (N>32,000)
**CalcForces2 Performance vs. T/B: N=131072**

<table>
<thead>
<tr>
<th># Flop/pair</th>
<th>N</th>
<th>#threads</th>
<th>Time [sec]</th>
<th>GFlop/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>131072</td>
<td>32</td>
<td>2.140</td>
<td>160.581</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>64</td>
<td>1.071</td>
<td>320.724</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>96</td>
<td>0.793</td>
<td>433.199</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>128</td>
<td>0.687</td>
<td>500.198</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>160</td>
<td>0.669</td>
<td>513.393</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>192</td>
<td>0.660</td>
<td>520.360</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>224</td>
<td>0.658</td>
<td>522.323</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>256</td>
<td>0.664</td>
<td>517.099</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>288</td>
<td>0.663</td>
<td>518.002</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>320</td>
<td>0.663</td>
<td>518.357</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>352</td>
<td>0.657</td>
<td>523.149</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>384</td>
<td>0.652</td>
<td>526.684</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>416</td>
<td>0.687</td>
<td>499.984</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>448</td>
<td>0.663</td>
<td>518.558</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>480</td>
<td>0.691</td>
<td>497.553</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>512</td>
<td>0.680</td>
<td>505.200</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>768</td>
<td>0.680</td>
<td>505.027</td>
</tr>
<tr>
<td>20</td>
<td>131072</td>
<td>1024</td>
<td>0.675</td>
<td>508.786</td>
</tr>
</tbody>
</table>

Large problems get good speed over a wide range of threadsPerBlock.
CalcForces2 Performance vs. T/B & N

Performance:
- Increases with N
- ~Constant for T/B > 100, N > 60,000
- Decreases with T/B for small problems

Best Speed:
- ~522 GFlop/s for 100 < T/B < 400, N > 60,000
Performance vs. Kernel Code & T/B

- N=141,072
- Impact of Memory Hierarchy
Performance vs. Kernel Code & N

T/B=320

Performance increases with size

CalcForces1
~4 times faster

CalcForces2
10-20% faster
Timer Synchronization

cudaEvent_t start, stop;
float elapsedTime;
cudaEventCreate( &start);
cudaEventCreate( &stop);

cudaEventRecord( start, 0);
Do_whatever<<<blocks,threads>>>(…);
cudaEventRecord( stop, 0);
cudaEventSynchronize( stop);
cudaEventElapsedTime( &elapsedTime, start, stop);

Prior to cudaEventSyncronize, host was just queuing up work for GPU

Without synchronization, host might get values from device before ANY work was done on GPU
Host-Device Synchronization

Need for timer synchronization illustrates an important CUDA run time feature:

• Most CUDA calls on host just “queue” work for GPU

• Host and Device run asynchronously
Asynchronously Running GPU: Valuable Impact on Performance

• Host can generate work for device without waiting for result or synchronizing with GPU.

• Avoids hand-shake delay or system interrupt
  – would lose a time slice (~1-10ms)

• Host can generate work for device in small pieces

• Only way the modular codes can perform well
  • Example: diffusion step: ~6 flop per cell
  • If loose a time slice (~1 ms) & 1 million cells
  → Would limit performance to 6 Gflop/s
Summary

• Can get > 500 Gflop/sec on a simple yet compute and memory intensive calculation
• Better performance on larger problem sizes
• Thread hierarchy is important
  – For GTX 480 & this code
  ~320 threads per block is optimal
• Memory hierarchy is important
  – Maximize use of thread local variables
  – Minimize traffic to global memory
Reading and Resources

• NVIDIA C CUDA Programming Guide


Or just search for the title (above)

Contact us: help@msi.umn.edu
Hands-On Exercises

• **Code**: calcForce.cu
  - In your NVIDIA_GPU_Computing_SDK/C/src directory:
  - `cp -r ~porterd0/calcForce .`
  - `cd calcForce ; see README file`

• **Explore performance vs.**:
  - threadsPerBlock
  - Problem size N
  - Code version

• **Restructure code to test effects on performance**
  - Do force calculation terms out of order
  - Examine code for further optimization

• **Implement full N-Body code with time step**