The Art of GPU Performance

Threading & Memory Hierarchy

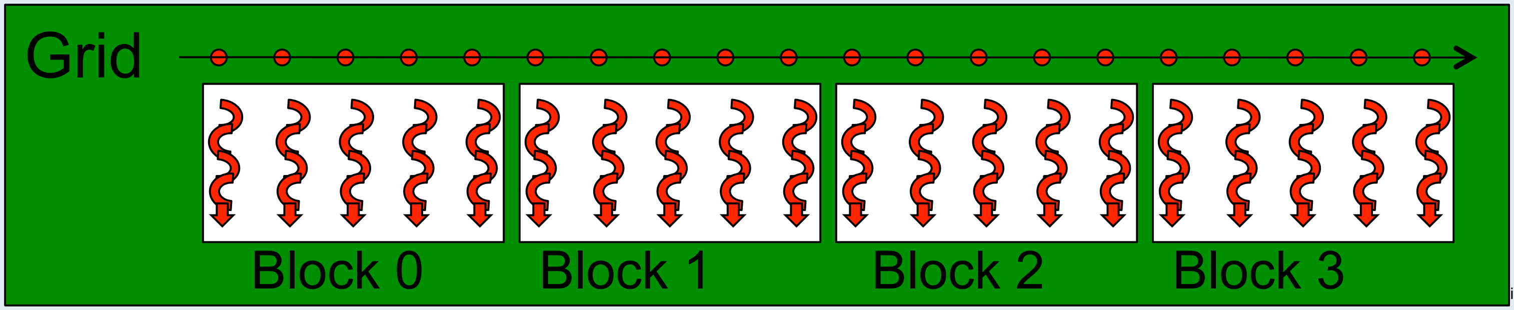
David Porter

“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

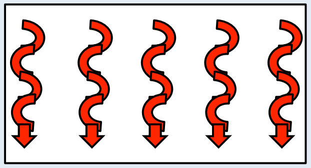


Example:
BPG=4
TPB=5

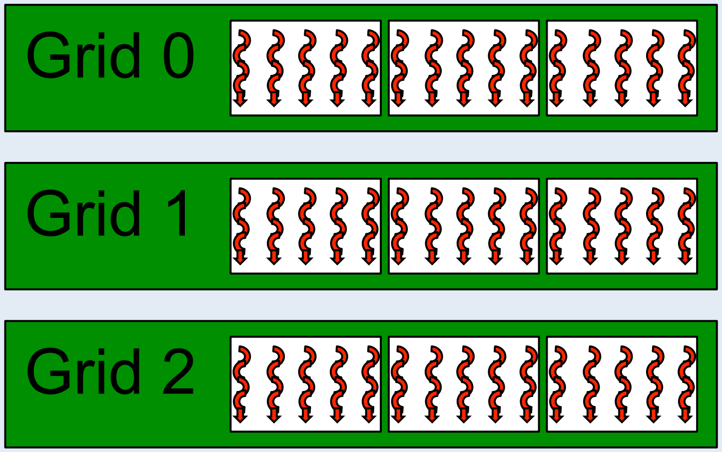
CUDA Memory Hierarchy on GPU

- 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



nVidia GeForce GTX 480

Multiprocessors	15
CUDA CORES/MP	32
Total CUDA Cores	480
Max threads per block	1024
Warp size	32
GPU Clock Speed	1.40 GHz
Memory Clock rate	1848.00 Mhz
Memory Bus Width	384-bit
Total global memory:	1536 MBytes
L2 Cache Size	786432 bytes
Constant memory	65536 bytes
Shared memory per block	49152 bytes
Registers per block	32768

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

Headers & Data Types

```
#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

Run Parameters & Timers

```
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);
```

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

For single precision, memory of each array is: 12*N bytes

Generate/Initialize Data

```
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

```
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

Cleanup

```
  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

```
__global__ void CalcForces0(const float3* pos, float3* 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, thredIdx.x provided
Value of i is unique on each thread and runs from 0 to N-1

Kernel Code CalcForces0: Work

```
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

Operation	Number
+ or -	9
* or /	9
rsqrtf	2
Total	20

`#define flop_per_pair 20`

Reads/Writes to force[i]
→ Traffic to global mem

CalcForces0 Performance vs. Size: T/B=320

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

- 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

CalcForces1: Work

```
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

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

Factor of ~4 speedup over CalcForce0 code

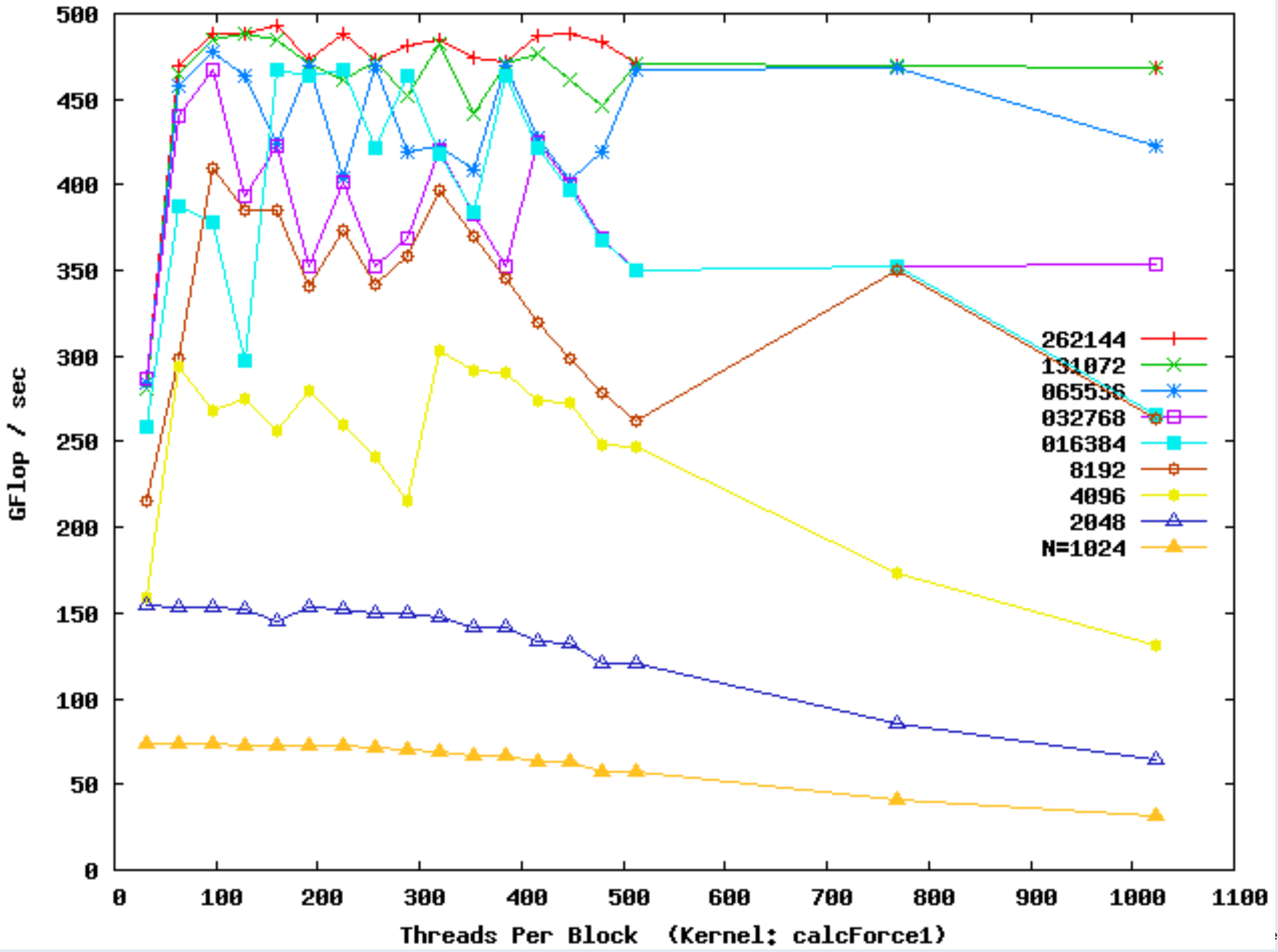
Performance now dramatically increases with problem size

CalcForces1 Performance vs.T/B: N=131072

For large N, performance is weakly dependent on threadsPerBlock when T/B> 32

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

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

```
__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

CalcForces2: Thread Evaluates fore[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

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

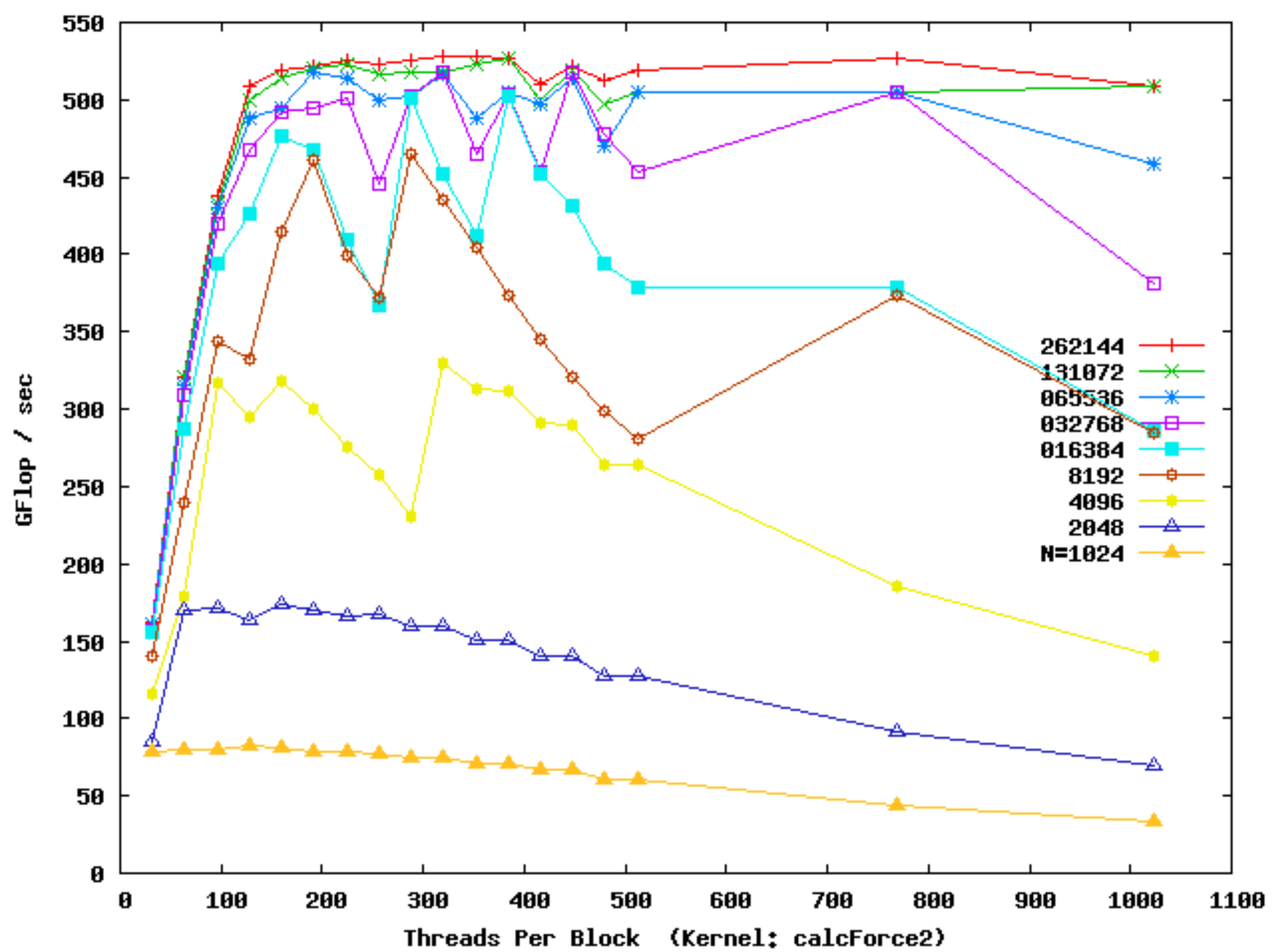
Over 500 GFlop/sec on large problem sizes (N>32,000)

CalcForces2 Performance vs.T/B: N=131072

Large problems get good speed over a wide range of threadsPerBlock

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

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

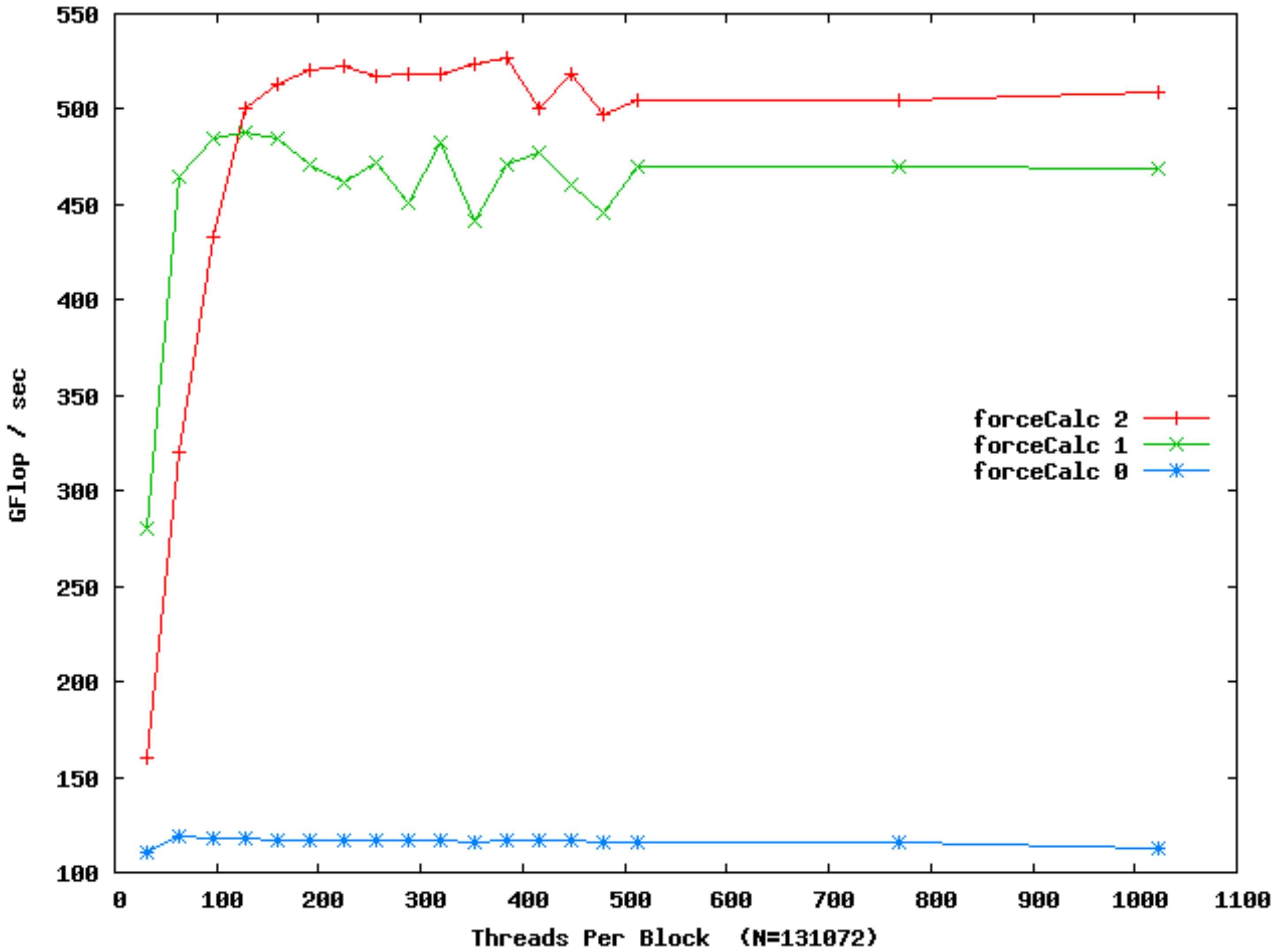
Best Speed:

~522 GFlop/s

100 < T/B < 400

N > 60,000

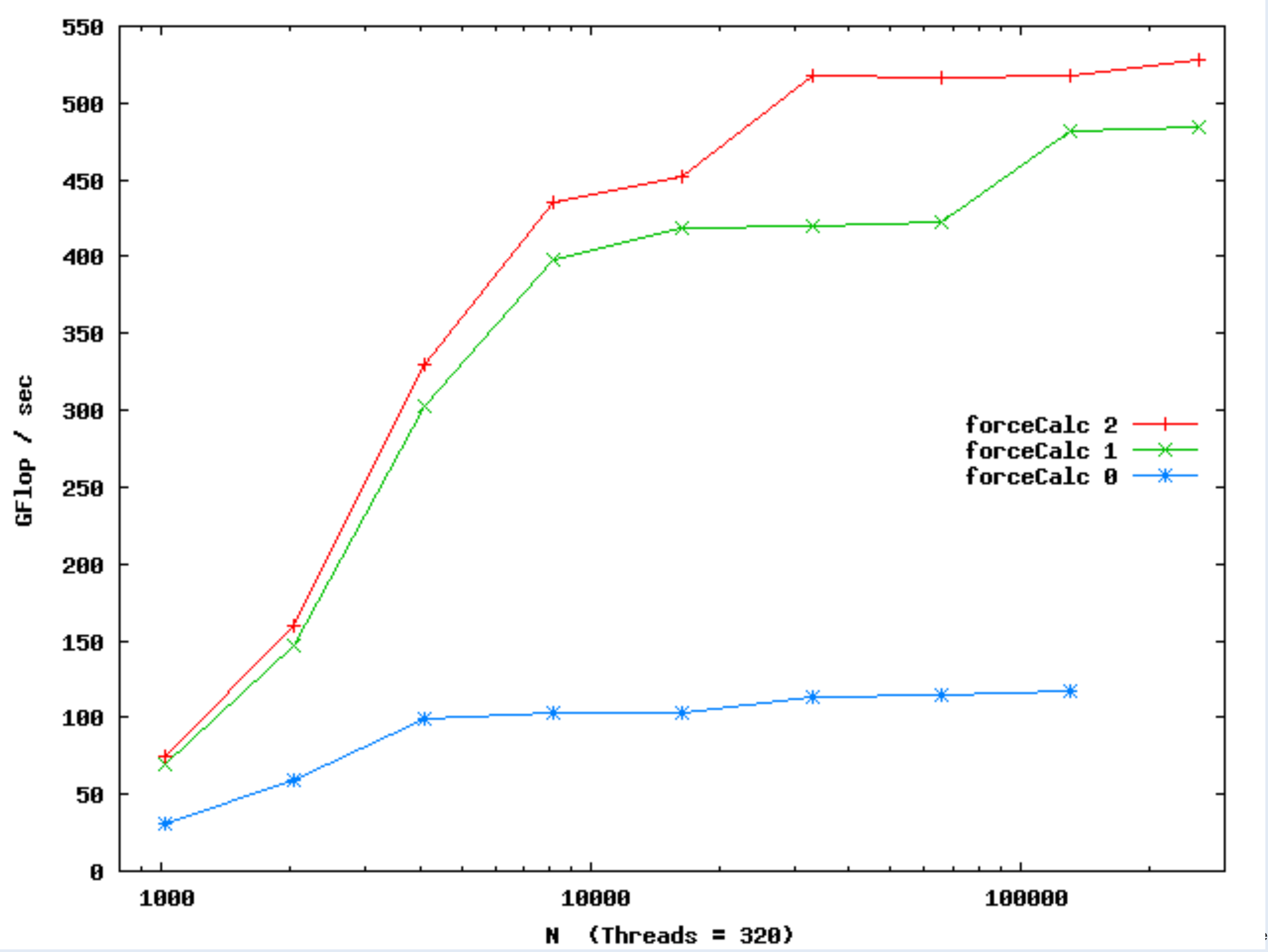
Performance vs. Kernel Code & T/B



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

  http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf

  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 –rp ~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**