Lecture 6

CUDA
Performance Programming
Announcements

• Forge: follow instructions to run the provided CUDA program, including batch
  cseweb.ucsd.edu/classes/wi12/cse260-a/env.html#Forge

• A2 has been posted
Today’s lecture

• Core dump for A1
  ☐ Performance Programming of Aliev-Panfilov
• In class exercises
• Fermi hardware
• CUDA
• A2
Performance programming: Aliev-Panfilov

- ./apf -n 1000 -t 5.0

Lilliput: 2.4, 4.7, 6.7 Gflops nt=(1,2,4); 1.3GF [serial]

Triton.sdsc.edu: 1.4, 2.8, 5.6, 11 Gflops nt=(1,2,4, 8); 1.3GF [serial]

```c
#pragma omp parallel for
for (j=1; j<=m+1; j++){
    _DOUBLE_ *RR = &R[j][1];
    _DOUBLE_ *EE = &E[j][1];
    #pragma ivdep
    for (i=1; i<=n+1; i++, EE++, RR++) {
        EE[0] = E_p[j][i]+α*(E_p[j][i+1]+E_p[j][i-1]-4*E_p[j][i]+E_p[j+1][i]
        +E_p[j-1][i]);
        EE[0] += -dt*(kk*EE[0]*(EE[0]-a)*(EE[0]-1)+EE[0]*RR[0]);
        RR[0] += dt*(ε+M1* RR[0]/( EE[0]+M2))*(-RR[0]-kk*EE[0]*(EE[0]-
        b-1));
    }
}
```
Iteration to thread mapping

#pragma omp parallel shared(N, iters) private(i)
#pragma omp for
for (i = 0; i < N; i++)
    iters[i] = omp_get_thread_num();

N = 9, # of openMP threads = 3
0 0 0 1 1 1 2 2 2

N = 16, # of openMP threads = 4, schedule(static,2)
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

N = 9: 0 0 1 1 2 2 0 0 1

N = 16, # of openMP threads = 4, schedule(dynamic,2)
3 3 0 0 1 1 2 2 3 3 3 3 3 3 3 3
2 2 3 3 0 0 1 1 2 2 2 2 2 2 2 2
Removing data dependencies

- B initially: \[0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\]
- B on 1 thread: \[0\ 9\ 10\ 11\ 12\ 21\ 22\ 23\]
- B on 2 threads: \[0\ 17\ 18\ 19\ 12\ 13\ 14\ 15\]
- **Claim:** we can split into 2 loops so that each loop parallelizes, and the result is correct

```plaintext
for i = 1 to N
    B[i] += B[N-i];
```
Splitting a loop

• For iterations \( i = N/2 + 1 \) to \( N \), \( B[200-i] \) reference newly computed data
• All others reference “old” data
• \( B \) initially: 0 1 2 3 4 5 6 7
• Correct result: 0 9 10 11 12 21 22 23

for \( i = 1 \) to \( N/2 \)
\[
B[i] += B[N-i];
\]
for \( i = N/2 + 1 \) to \( N \)
\[
B[i] += B[N-i];
\]

Today’s lecture

• Core dump for A1
  ◆ Performance Programming of Aliev Panfilov
• In class exercises
• Fermi Hardware
• CUDA
• A2
Heterogeneous processing
CUDA

• Programming environment + C extensions
• Under control of the *host*, run a sequence of multi-threaded GPU kernels on the *device*
• Extremely lightweight virtualized threads
Fermi Testbeds

- All the cseclass machines have Fermi processors
  - Private L1 Cache/Shared Memory (64KB/Vector unit), Shared L2 Cache (768 KB)
- CSEClass 01, 02: GeForce GTX 580 [Capability 2.0, GF100]
  - 15 Vector units @ 32 cores/unit (480 cores), 4 SFUs
  - 1.25 GB device memory
- CSEClass 03-07: GeForce GTX 460 Capability 2.1, GF104]
  - 7 Vector units @ 48 cores (384 total cores), 8 SFUs
  - 1.0 GB device memory

www.anandtech.com/show/3809/nvidias-geforce-gtx-460-the-200-king/2
CSEClass 01, 02: GeForce GTX 580 [2.0, GF100]
15 Vector units @ 32 cores/unit (480 cores), 4 SFUs
1.25 GB device memory

CSEClass 03-07: GeForce GTX 460 [2.1, GF104]
7 Vector units @ 48 cores (384 total cores), 8 SFUs
1.0 GB device memory

Forge: Tesla M2070 [2.0, GF100]
6 devices per node
14 Vector units @ 32 cores (448 total cores), 4 SFUs
6 GB device memory + ECC (5.25GB usable)
SP MAD: 1030.4 Gflops, DP FMA: 515.2
Fermi Improvements

- Larger vector units, more total cores
- Higher peak double precision performance
- L1 Cache, configurable as $\frac{3}{4}$ L1 or SM, $\frac{1}{4}$ SM or L1
- L2 Cache
- Dual thread schedulers
- Concurrent kernel execution (some models)
- Reduced kernel launch overhead (25 $\mu$s)
- Improved predication
Vector units

• Each vector unit
  • 32 CUDA cores for integer and floating-point arithmetic
  • 4 special function units for Single Precision transcendental
  • FMA without truncation (32 or 64 bits)

• For devices of compute capability 2.1
  • 48 CUDA cores for arithmetic operations
  • 8 special function units for single-precision

• CUDA C Programming Guide Version 4.0, §G.3
CUDA language extensions

- Type qualifiers to declare device kernel functions
  `__global__` void matrixMul( …)
- Kernel launch syntax
  `matrixMul<<grid, threads >>>(…)
- Keywords
  `blockIdx`, `threadIdx`, `blockDim`, `gridDim`
- Runtime, e.g. storage allocation
  `cudaMalloc`, `cudaFree`, `cudaMemcpy`
Coding example – Increment Array

Serial Code

```c
void incrementArrayOnHost(float *a, int N){
    int i;
    for (i=0; i < N; i++) a[i] = a[i]+1.f;
}
```

```c
#include <cuda.h>
__global__ void incrementOnDevice(float *a, int N) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx]+1.f;
}

incrementOnDevice <<< nBlocks, blockSize >>> (a_d, N);
```

Rob Farber, Dr Dobb’s Journal
Managing memory

float *a_h, *b_h;       // pointers to host memory
float *a_d;             // pointer to device memory

cudaMalloc((void **) &a_d, size);

for (i=0; i<N; i++) a_h[i] = (float)i;  // init host data

cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
Computing and returning result

int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
incrementOnDevice <<< nBlocks, bSize >>> (a_d, N);

// Retrieve result from device and store in b_h
cudaMemcpy(b_h, a_d, sizeof(float)*N,
cudaMemcpyDeviceToHost);

// check results
for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);

// cleanup
free(a_h); free(b_h);
cudaFree(a_d);
Experiments - increment benchmark

- Total time: timing taken from the host, includes copying data to the device
- Device only: time taken on device only

<table>
<thead>
<tr>
<th>Reps</th>
<th>Device time</th>
<th>Kernel launch + data xfer</th>
<th>Host</th>
<th>Sine function (Host)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8.5</td>
<td>29</td>
<td>77</td>
<td>16</td>
</tr>
<tr>
<td>100</td>
<td>83</td>
<td>100</td>
<td>770</td>
<td>103</td>
</tr>
<tr>
<td>1000</td>
<td>830</td>
<td>850</td>
<td>7700</td>
<td>23.6 sec</td>
</tr>
<tr>
<td>10000</td>
<td>8300</td>
<td>8300</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(N = 8388480, \) block size = 128, times in milliseconds, Lilliput

Measuring performance

• Two ways
  ♦ Use an ordinary timer, e.g. gettimeofday()
  ♦ Use Cuda events/elapsed time (#ifdef CUDA_TIMER)

• See incrArray

• Note that kernel invocation is asynchronous

    cudaThreadSynchronize();
    double t_device_compute = -getTime();
    incr<<< nBlocks, bSize >>> (a_d, N);
    cudaThreadSynchronize();
    t_device_compute += getTime();
Cuda error: Can't run kernel: invalid device function.

• Cuda can silently fail, you can observe misleading performance
• E.g. if you specify an invalid grid / thread block dimensions
• Note: the last error can be cleared by successive kernel calls, so check frequently

    cudaMalloc((void **) &a_d, size);
    checkCUDAError("Unable to allocate storage on the device");

• Consult checkCUDAError() in utils.cu (incrArr)
• What about asynchronous calls?
• cf CUDA Programming Guide, “Error Handling”
Getting information about the binary

- Compiler will report a kernel’s register usage along with that of local, shared and constant memory

```c
--ptxas-options=-v
incrementArrays (float *a, int N)
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx]+1.f;
```

ptxas info : Compiling entry function
'_Z22incrementArrayOnDevicePfii' for 'sm_13'
ptxas info : Used 4 registers, 16+16 bytes smem, 4 bytes cmem[1]
Warp scheduling

• Blocks are divided into *warps* of 32 (SIMD) threads which are …. 
• … subdivided into schedulable units: 16 threads (→ 32 on Fermi)
• Scheduler finds an eligible warp: all operands are ready
  - Scoreboarding, priorities
  - Zero overhead
• Branches serialize execution in warp
• To execute an instruction for all threads of a warp, the warp scheduler must issue the instruction over
  - 4 clock cycles for an integer or single-precision floating-point arithmetic instruction [throughput = 8]
  - 32 cycles for double-precision [1]
  - 16 cycles for single-precision floating-point transcendental [2]
Instruction issue on Fermi

- Each vector unit
  - 32 CUDA cores for integer and floating-point arithmetic
  - 4 special function units for Single Precision transcendentals
- 2 Warp schedulers: each scheduler issues: 1 (2) instructions for capability 2.0 (2.1)
- One scheduler in charge of odd warp IDs, the other even warp IDs
- Only 1 scheduler can issue a double-precision floating-point instruction at a time
- Warp scheduler can issue an instruction to $\frac{1}{2}$ the CUDA cores
- Scheduler must issue the instruction over 2 clock cycles for an integer or floating-point arithmetic instruction
Dynamic behavior – resource utilization

• Each vector core (SM): 1024 thread slots and 8 block slots
• Hardware partitions slots into blocks at run time, accommodates different processing capacities
• Registers are split dynamically across all blocks assigned to the vector core
• A register is private to a single thread within a single block
Constraints

• SM
  - Up to 8 resident blocks
  - Not more than 1024 threads
  - Up to 32 warps

• All threads in a warp execute the same instruction
  - All branches followed
  - Instructions disabled
  - Divergence, serialization

• Blocks
  - ≤ 512 threads
  - Registers subdivided over threads
  - Synchronization among all threads in the block, only within block
Occupancy

- A minimum number of warps needed to hide memory latency
- Occupancy: \( \frac{\# \text{ active warps}}{\max \# \text{ warps supported by vector unit}} \)
- Limited by vector unit resources
  - Amount of shared memory
  - Number of registers
  - Maximum number of threads

- Consider a kernel (16x16 block size)
  - Shared memory/block = 2648 bytes
  - Reg/thread=38 \([38 \times 256 = 9728 < 16k]\)
  - \# available registers is the limiting factor

- Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - Register consumption
Programming issues

• Branches serialize execution within a warp
• Registers dynamically partitioned across each block in a Streaming Multiprocessor
• Bound to and only accessible from their thread until the block finishes execution
• Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - Register consumption
  - Scheduling: hide latency
• Shared memory and registers do not persist across kernel invocations
CUDA Runtime

- CUDA includes a runtime library
- No explicit initialization procedure
- *Cuda C Programming Guide, §3.2*
  - Initializes the first time a runtime function is called
  - Initialized by function *other than* functions from the *device* and *version management* sections of the reference manual
  - e.g. `cudaGetLastError()`, `cudaEventCreate()`, `cudaMalloc()`
  - NOT `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`


- Beware of erroneous timings
- Careful when interpreting the error code from the first call into the runtime
- CUDA Context: like a CPU process (see §3.3.1)
  - Created as part of initialization
  - Host thread that initializes the run time is the only host thread with access to device memory allocated within that context, events, too
  (see CUDA timer in `incrArr`)
Today’s lecture

• Core dump for A1
  - Performance Programming of Aliev Panfilov
• In class exercises
• Fermi Hardware
• CUDA
• A2
Aliev-Panfilov implementation under CUDA

\[ E^t[i,j] = E^{t-1}[i,j] + \alpha(E^{t-1}[i+1,j] + E^{t-1}[i-1,j] + E^{t-1}[i,j+1] + E^{t-1}[i,j-1] - 4E^{t-1}[i,j]) \]
Naïve Aliev-Panfilov Code

- All array references go through device memory
- 
  .apf -n 6144 -t 0.04, 16x16 thread blocks
  - Lilliput, cseclass01, cseclass05
  - SP: 22, 73, 34 GFlops [Triton, 32 cores, MPI: 85GF]
  - DP: 13, 45, 20 GFlops (19GF n=8192) [Triton: 48 GF]

```c
#define E'[i,j] E_prev[(j+1)*(m+3) + (i+1)]
I = blockIdx.y*blockDim.y + threadIdx.y;
J = blockIdx.x*blockDim.x + threadIdx.x;
if ((I <= n) && (J <= m) )
```

```c
for (j=1; j<= m+1; j++)
    for (i=1; i<= n+1; i++)
        E[j][i] = E'[j][i] + \alpha*(E’ [j][i-1]+E’ [j][i+1] + E’ [j-1][i]+E’ [j+1][i] - 4*E’ [j][i]);
```
Assignment

• Starting from a basic (unoptimized) CUDA implementation
• Optimize matrix multiplication for the GPU
  ♦ Fermi and 200 series
• Next, implement matrix multiplication in MPI
  ♦ Lilliput → Trestles, if time Kraken
• You’ll be given an efficient (blocked for cache) serial implementation

• Goals
  ♦ Understand differences between Fermi and 200 series GPUs
  ♦ Comparative performance across platforms: how many MPI cores needed to deliver comparable performance to the GPU?
Fin