Lecture 8

SIMD processing and vectorization

Stream processing
Announcements

• Assignment return and comments
• Bug in residual computation, new code posted
• Feedback for the Projects
  – Make an appointment to see me
• Lincoln Accounts
• A3
  – Implement the iterative solver with MPI or GPU
• Feedback for A1
Sampling Errors

TRITON

LAPTOP
Accounting for cache size

<table>
<thead>
<tr>
<th>Matrix size (n)</th>
<th>Repetitions</th>
<th>Number of subblocks (N)</th>
<th>Time (sec)</th>
<th>Size of subblock (n/N)</th>
<th>Memory required to store 3 matrices of size n/N x n/N, in kB (MR)</th>
<th>Relation to cache size</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>1</td>
<td></td>
<td>692.66</td>
<td>1024</td>
<td>24576 kB</td>
<td>L3 &lt; MR</td>
<td>In this case 3 matrices don’t fit into L3-cache, so we see the worst performance.</td>
</tr>
<tr>
<td>8</td>
<td>185.29</td>
<td>512</td>
<td>6144 kB</td>
<td>L2 &lt; MR &lt; L3</td>
<td>3 matrices fit into L3 cache, performance is 4 times better than in above case.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>173.54</td>
<td>256</td>
<td>1536 kB</td>
<td>L2 &lt; MR &lt; L3</td>
<td>Still 3 matrices fit into L3, but doesn’t fit into L2, so performance is almost the same.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>134.99</td>
<td>128</td>
<td>384 kB</td>
<td>L2 &lt; MR &lt; L3</td>
<td>In this case performance is slightly better because 1-1.5 matrices fit into L2 cache, and the rest fits into L3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>90.59</td>
<td>64</td>
<td>96 kB</td>
<td>L1 &lt; MR &lt; L2</td>
<td>Matrices fit into L1/L2 cache, so performance is relatively high.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>89.02</td>
<td>32</td>
<td>24 kB</td>
<td>MR &lt; L1</td>
<td>All submatrices fit into L1 cache, so we should see the best performance here, but it’s not so high because we have many many such subblocks (overhead matters).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>118.58</td>
<td>16</td>
<td>6 kB</td>
<td>MR &lt; L1</td>
<td>Performance degrades due to growing number of small subblocks.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| S12 | 180.71 | 8 | 1.5 kB | MR < L1 | Performance degrades, because CPU resources are underutilized, we have lots of free cache space and we process small pieces of data, and during each phase we have to copy data from big matrices into subblocks A, B, C, then compute, and the copy values back to big matrices (this is overhead). |

Dependence of time on partitioning

- R1 in L1
- R1 in L2
- R1 in L3
- doesn’t fit in the L2

<table>
<thead>
<tr>
<th>Size of subblock (n/N)</th>
<th>Number of subblocks (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-cache</td>
<td>32 kB</td>
</tr>
<tr>
<td>L2-cache</td>
<td>256 kB</td>
</tr>
<tr>
<td>L3-cache</td>
<td>8192 kB</td>
</tr>
</tbody>
</table>
Further optimizations

Sub-blocks
Loop unrolling

for (int i = 0; i < n; i++)
for (int k = 0; k < n; k++) {
    // Reduce references to A
    const double a_ik = A[i][k];
    for (int j = 0; j < n; j++)
        C[i][j] += a_ik * B[k][j];
}
Road Map

- Vector processing (SIMD)
- Introduction to Stream Processing
Cray-1 Block diagram

memory bank cycle 50 ns  processor cycle 12.5 ns (80MHz)
Streaming SIMD Extensions

- en.wikipedia.org/wiki/Streaming_SIMD_Extensions
- SSE (SSE4 on Triton’s Nehalems), Altivec
- Short vectors: 128 bits (256 bits coming)
Fused Multiply/Add

\[ r[0:3] = c[0:3] + a[0:3]*b[0:3] \]

Courtesy of Mercury Computer Systems, Inc.
How do we use the SSE instructions?

• Low level: assembly language or libraries
• Higher level: a vectorizing compiler
  
  ```
  pgcc -fastsse -Minfo=all prog.c
  float a[N], b[N], c[N];
  for (int i=0; i<N; i++)
    a[i] = b[i] + c[i];
  ```

  13, Generated an alternate loop for the inner loop
  Generated vector sse code for inner loop
  Generated vector sse code for inner loop

• If value of N is **not** known at compile time, compiler generates code for different cases
  
  13, Generated 4 alternate loops for the inner loop
  Generated vector sse code for inner loop
  ...
  Generated vector sse code for inner loop
How does non-vectorized code compare?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler

```c
#pragma novector
for (int i=0; i<N; i++) // N = 2048 * 1024
    a[i] = b[i] + c[i];
```

With vectorization : 7.693 sec.
Without vectorization : 10.17 sec.

- Double precision
  With vectorization : 11.88 sec.
  Without vectorization : 11.85 sec.
How does the vectorizer work?

- Transformed code
  
  ```c
  for (i = 0; i < 1024; i+=4)
      a[i:i+3] = b[i:i+3] + c[i:i+3];
  ```

- Vector instructions
  
  ```c
  for (i = 0; i < 1024; i+=4){
      vB = vec_ld( &b[i] );
      vC = vec_ld( &c[i] );
      vA = vec_add( vB, vC );
      vec_st( vA, &a[i] );
  }
  ```
What prevents vectorization

• Data dependencies
  for (int i = 1; i < N; i++)
    b[i] = b[i-1] + 2;

4, Loop not vectorized: data dependency

Loop not vectorized: data dependency

• Inner loops only
  for(int j=0; j< reps; j++)
    for (int i=0; i<N; i++)
      a[i] = b[i] + c[i];

9, Generated vector sse code for inner loop
What prevents vectorization

- Interrupted flow out of the loop
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
    if (maxval > 1000.0) break;
  }

6. Loop not vectorized/parallelized: multiple exits

- This loop will vectorize
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
  }

Dealing with spurious dependencies

• Tell the compiler to ignore the dependence

212: for (i=1; i<nx1; i++)
213: for (j=1; j<ny1; j++)
214: #pragma ivdep
215: for (k=1; k<nz1; k++)

Un[i][j][k] = c * (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] +
U[i][j+1][k] + U[i][j][k-1] + U[i][j][k+1] -
c2*b[i-1][j-1][k-1]);

212, Loop not vectorized/parallelized: too deeply nested
215, Unrolled inner loop 4 times!

• Without the pragma (What is causing this to happen?)

214, Loop not vectorized: data dependency
Loop not vectorized: data dependency
Computing with Graphical Processing Units (GPUs)
Technological disruption

• New capabilities
• Changes the common wisdom for solving a problem including the implementation

Cray-1, 1976, 240 Megaﬂops

ASCI Red, 1997, 1Tflop

Intel Teraﬂop on a chip 2007

Nvidia Tesla, 4.14 Tflops
NVIDIA GeForce GTX 280

- Hierarchically organized clusters of streaming multiprocessors
  - 240 cores @ 1.296 GHz
  - Peak performance 933.12 Gflops/s
- SIMT parallelism
- 1 GB “device” memory (frame buffer)
- 512 bit memory interface @ 132 GB/s

GTX 280: 1.4B transistors
Intel Penryn: 410M (dual core)
Streaming processing cluster

- Each cluster contains 3 streaming multiprocessors
- Each multiprocessor contains 8 cores that share a local memory
- 3 flops/core/cyc * 8 cores/SM * 3 SM/cluster * 10 clusters = 720 flops/cyc
- @ 1.296 Ghz: 933 GFLOPS/EC
- Double precision is ~10× slower than floating point
Streaming Multiprocessor

- 8 cores (Streaming Processors)
  - Each core contains 1 fused multiply adder (single precision), truncates intermediate result
  - May complete 1FMA + 1 multiply per cycle = 3 flops / cycle
  - Share one 64-bit fused multiply adder
  - 2 Super Function Units (SFUs), each contains 2 fused multiply-adders
- 16 KB shared memory + 16K registers
CUDA

- Programming environment with extensions to C
- Model: Execution on the CPU, run a sequence of multi-threaded kernels on the “device” (GPU)
- Threads are extremely lightweight and virtualized
Hierarchical Thread Organization

- Grid > Block > Thread
- Thread Blocks
  - Cooperate, synchronize, with access fast on-chip shared memory
  - Threads in different blocks communicate only through slow global memory
  - Blocks within a grid are virtualized, too
  - May configure number of threads in a block and the number of blocks
- Threads assigned to SM in units of blocks, up to 8 for each SM
- Compiler re-arranges loads to hide latencies

KernelA<<<2,3>,<3,5>>>
Coding example – Increment Array

• Rob Farber, Dr Dobb’s Journal

Serial Code

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

Rob Farber, Dr Dobb’s Journal
Increment Array Kernel

#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);
Managing memory

```c
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);

int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
```
Transferring the Data

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);
Memory Hierarchy

<table>
<thead>
<tr>
<th>Name</th>
<th>Latency (cycles)</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Local</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Constant</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Texture</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Shared</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>Register</td>
<td>1</td>
<td>--</td>
</tr>
</tbody>
</table>

Courtesy David Kirk/NVIDIA and Wen-mei Hwu/UIUC
Warp scheduling (8800)

- Blocks contain multiple warps, a group of SIMD threads
- Half-warps are the unit of scheduling (16 threads currently)
- Hardware scheduled warps hide latency (zero overhead)
  - Scheduler looks for an eligible warp with all operands ready
  - All threads in the warp execute the same instruction
  - All branches followed: serialization, instructions can be disabled
- Many warps need to hide memory latency
- Registers shared by all threads in a block

Courtesy David Kirk/NVIDIA
Wen-mei Hwu/UIUC
Registers and Shared Memory

• Registers are dynamically partitioned across each block in an Streaming Multiprocessor (16K in GTX-285)
• Shared memory: 16K per SM
• Bound to and only accessible from their thread until the block finishes execution
• Can choose between more blocks with fewer threads or more threads with fewer blocks
Nested Granularity (8800)

Figure 6: Nested granularity levels: thread (a), cooperative thread array (b), and grid (c). These have corresponding memory-sharing levels: local per-thread, shared per-CTA, and global per-application.
LANGUAGE EXTENSIONS

- Superset of C - .cu files compile to C files for both the device and the host
- New keywords (__global__, __device__, __shared__, __syncthreads())
- New types (dim3, uint[1..4], float[1..4], …)
- New built-in variables (gridDim, blockDim, blockIdx, threadIdx)
- __global__ and __device__ functions have limitations since they run on the device
  - No recursion
  - No closures (static function variables)
  - No varargs
Sequenced kernels

Figure 7. CUDA program sequence of kernel A followed by kernel B on a heterogeneous CPU–GPU system.
CUDA Coding Example

Figure 8. Serial C (a) and CUDA C (b) examples of programs that add arrays.
Programming issues

• Branches serialize execution within a warp
• Registers dynamically partitioned across each block in a Streaming Multiprocessor
• 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
MATRIX MULTIPLICATION EXAMPLE

• If each Block has 16×16 threads and each thread uses 10 registers, how many threads can run on each SM?
  – Each Block requires 10*256 = 2560 registers
  – 8192 = 3 * 2560 + change
  – So, three blocks can run on an SM as far as registers are concerned

• How about if each thread increases the use of registers by 1?
  – Each Block now requires 11*256 = 2816 registers
  – 8192 < 2816 *3
  – Only two Blocks can run on an SM, 1/3 reduction of parallelism!!!

* Slide from David Kirk/NVIDIA and Wen-mei W. Hwu, 2007, ECE 498AL, UIUC
Example – Array Increment

```c
float *a_h, *b_h;       // pointers to host memory
float *a_d;             // pointer to device memory
int i;
size_t size = N * sizeof(float);
// allocate arrays on host
a_h = (float*)malloc(size);
b_h = (float*)malloc(size);
// allocate array on device
cudaMalloc((void **)&a_d, size);
// initialization of host data
for (i = 0; i < N; i ++)
    a_h[i] = (float)i;
// copy data from host to device
cudaMemcpy(a_d, a_h, sizeof(float)*N,
    cudaMemcpyHostToDevice);
```
int blockSize = 1;
int nBlocks = N / blockSize + (N % blockSize == 0 ? 0 : 1);

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

free(a_h); free(b_h); cudaFree(a_d);

__global__ void incrementArrayOnDevice(float *a, int N)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + 1.f;
}
OUTLINE

• Introduction to CUDA
• Performance Concerns
BRANCHES

• All the threads in a warp should follow the same control path
  – Different control paths are serialized
• Avoid divergence when predicates are a function of threadId
  – BAD
    • if (threadId < 2) { }
  – GOOD
    • if (threadId / WARP_SIZE < 2) { }
PREDICATES

• Save branch divergences – good for if-else

• `<p1> ADD r1, r2, 0`
  – If `p1` is true, perform the add, otherwise no-op

  BNE r1, 0, L1
  ADD r2, r2, r3
  JMP L2
  L1: ADD r2, r2, r4
  L2: …

  \[\text{\textit{\iff r1=0}}} \text{ ADD r2, r2, r3} \]
  \[\text{\textit{\iff r1\neq0}}} \text{ ADD r2, r2, r4} \]

• No divergence
• Each thread executes both adds (only one saved)
BANK CONFLICTS

• Analogous to cache-alignment issues
• Shared memory is stored in different banks
• Shared memory is visible to all threads within a block (r/w access)
• 16 banks per SM on the G80 (bank = address % 16)
• Each bank can serve one address per cycle
  – Multiple accesses to a bank at once result in a conflict
    • Conflicts are serialized!
  – If each thread reads the same address, broadcast used
BANK CONFLICTS (CONT.)

• Since threads are scheduled within their warp, watch strides
• Sequential access won’t duplicate banks
  – Good
    • int a = sharedArray[threadId.x];
  – Bad
    • int a = sharedArray[threadId.x * 2];
• As long as the multiplier of threadId.x is not a multiple of the number of banks, won’t have conflicts
  – sharedArray[threadId.x * s];
  – s should not be a multiplier of 16 (for the G80) – make it odd
BANK CONFLICTS (CONT.)

• Common patterns – loading from global memory to a shared array
  – Bad – 2 way conflict
    int i = threadId.x;
    shared[2 * i] = global[2 * i];
    shared[2 * i + 1] = global[2 * i + 1];
  – Good – load from subsequent blocks
    int i = threadId.x;
    shared[2 * i] = global[2 * i];
    shared[2*i+blockDim.x]=
      global[2*i+blockDim.x];
PARALLELISM

• Instruction-Level vs. Thread-Level
  – Compiler will re-arrange loads to hide latencies
  – Hardware will schedule threads during latencies
  – Can be better to have fewer threads
  – Use the occupancy calculator* for residency information

• Example (Kirk and Hwu, NVIDIA, UIUC)
  – 256-threads per block, 10 registers per thread
  – 4 independent instructions for each global memory load
  – 200 cycle global load latency
  – 3 Blocks can run on each SM
  – If a compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load - only 2 blocks can run on each SM
  – But then we only need 200 / (8 * 4) = 7 warps to tolerate the memory latency
  – Two blocks have 16 warps, so the performance can be higher!

RUN-TIME RESOURCES

• `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`
  – Return run-time info if want to auto-tune

• Usage counters
  – Maintained per warp
  – Need lots of blocks for accurate counts