Lecture 18

Bitonic Sort
Graphical Processing Units
Announcements

• Quiz
• In class problems
In class problem solving
Performance model

• Local additions: \( N/P - 1 \)
• Reduction: \((1+\alpha)(\log P - 1)\)
• \( T(N,P) = \frac{N}{P} + \alpha \log P \)
• Determine the largest problem that can be solved in time \( T = 10^4 \) time units on \( P=512 \) processors, \( \alpha = 10 \) time units, and addition costs one unit of time
• Consider \( T(512,N) \leq 10^4 \)
  \( \Rightarrow (N/512) + \alpha \log(512) \leq 10^4 \)
  \( \Rightarrow (N/512) + 90 \leq 10^4 \)
  \( \Rightarrow N \leq 5 \times 10^6 \) (approximately)
Time constrained scaling

- Sum N numbers on P processors
- Let N >> P
- Determine the largest problem that can be solved in time T=10^4 time units on 512 processors
- Let the startup time be $\alpha$ time units
- Let $\beta = 1/(\text{peak bandwidth})$
- Assume that $\beta = \text{time to perform one addition}$
Today’s lecture

• More irregular problems
• Vector processing
• Graphical Processing Units
Sparse Matrices
Sparse Matrices

- A matrix where knowledge about the location of the non-zeroes is useful
- Consider the a 5-point stencil

\[ u'[i,j] = (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i, j+1]) / 4 \]
Savings with Sparse Matrices

- 7 x 7 grid: 49 x 49 matrix
- Nonzeroes: 349, 30% of the space
- Flops: 2643, 6.7%
- Fill-in: a zero entry becomes nonzero
Web connectivity Matrix: 1M x 1M

1M x 1M submatrix of the web connectivity graph, constructed from an archive at the Stanford WebBase

3 nonzeroes/row
Circuit Simulation

Motorola Circuit

$170,998^2$
958,936 nonzeroes
.003% nonzeroes
5.6 nonzeroes/row

www.cise.ufl.edu/research/sparse/matrices/Hamm/scircuit.html
Irregular mesh: NASA Airfoil in 2D

Jim Demmel
Sparse Matrix Vector Multiplication

- Important kernel used in sparse linear algebra: \( y[i] += A[i,j] \times x[j] \)
- Many formats, common format for CPUs is Compressed Sparse Row (CSR)

Jim Demmel
Bitonic Sort
Bitonic sort

- Classic parallel sorting algorithm $O(\log^2 n)$ time on $n$ processors
- Also used in fast sorting on a GPU
- **Definition:** A *bitonic sequence* is a sequence of numbers $a_0, a_1 \ldots a_{n-1}$ with the following properties
  - There exists an index $i$ where $a_0 \leq a_1 \leq a_1 \ldots \leq a_i$ and $a_i \geq a_{i+1} \geq a_{i+1} \ldots \geq a_{n-1}$
  - We may cyclically shift the $a_k$ while maintaining this relationship

1,2,4,7,6,0

7,6,0,1,2,4
Merge property of a bitonic sequence

- We may merge two bitonic sequences in much the same way as we merge two monotonic sequences
Splitting property of bitonic sequences

- We can split a bitonic sequence $y$ into two bitonic sequences $L(y)$ and $R(y)$

$$L(y) = \langle \min \{a_0,a_{n/2}\}, \min \{a_1,a_{n/2+1}\}, \ldots, \min \{a_{n/2+1},a_{n-1}\} \rangle$$

$$R(y) = \langle \max \{a_0,a_{n/2}\}, \max \{a_1,a_{n/2+1}\}, \ldots, \max \{a_{n/2+1},a_{n-1}\} \rangle$$

- See the notes for a proof

All values in $L(y) < R(y)$

$L(y): 3 4 2 1$

$R(y): 7 5 8 9$
Sorting a bitonic sequence is easy

- Split the bitonic sequence $y$ into two bitonic subsequences $L(y)$ & $R(y)$
- Sort $L(y)$ and $R(y)$ recursively
- Merge the two sorted lists
  - Since all values in $L(y)$ are smaller than all values in $R(y)$ we don’t need to exchange values in $L(y)$ and $R(y)$
- When $|L(.)| < 3$, sorting is trivial
- We designate $S(n)$ to be sort on of an $n$-element bitonic sequence
Bitonic sort algorithm

- Create a bitonic sequence $y$ from an unsorted list
- Apply the previous algorithm to sort the bitonic sequence
- We need an algorithm to create the bitonic sequence $y$
Additional properties of bitonic sequences

- Any 2 element sequence is a bitonic sequence
- We can trivially construct a bitonic sequence from two monotonic sequences, one sorted in increasing order, the other in decreasing order
Inductive construction of the initial bitonic sequence

- Form matched pairs of 2-element bitonic sequences, pointing up and down \([B(2)]\)
- Trivially merge these into 4-element bitonic sequences
- Now form matched pairs of 4-element sequences \([B(4)]\)
- Apply \(S(4)\) to each sequence, sorting the first upward, the second downward
- Trivially merge into an 8-element bitonic sequence
- Continue until there is just one sequence
Implementing the bitonic sort algorithm

- Create a bitonic sequence $y$ from an unsorted list, $B(n)$
- Apply the previous algorithm to sort the bitonic sequence, $S(n)$
- We use comparators to re-order data
- We use a shuffle exchange network to form $L(y)$ and $R(y)$
  - This network shuffles an $n$-element sequence by interleaving $x_0$, $x_{n/2}$, $x_1$, $x_{n/2+1}$, ...
Comparators

• Given two values $x$ & $y$, produce two outputs

• For an increasing comparator, the output is $\min[x,y], \max[x,y]$

• For a decreasing comparator, the output is $\max[x,y], \min[x,y]$

\[ x \quad \text{Min}[x,y] \]
\[ y \quad \text{Max}[x,y] \]

\[ x \quad \text{Max}[x,y] \]
\[ y \quad \text{Min}[x,y] \]
Bitonic merging network

• Converts a bitonic sequence into a sorted sequence

From Introduction to Parallel Computing, V. Kumar et al, Benjamin Cummings, 1994
Bitonic conversion network

Converts an unordered sequence into a bitonic sequence

\[ B(4) = S(4) + S(2) \]

From Introduction to Parallel Computing, V. Kumar et al, Benjamin Cummings, 2003
Vector processing
Streaming SIMD Extensions

- [en.wikipedia.org/wiki/Streaming_SIMD_Extensions](en.wikipedia.org/wiki/Streaming_SIMD_Extensions)
- SSE (SSE4 on Intel Nehalem), Altivec
- Short vectors: 128 bits (256 bits coming)

![Diagram of Streaming SIMD Extensions](https://example.com/diagram.png)

16-byte boundaries

b[l] + c[l]

Courtesy of International Business Machines Corporation.
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

```c
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];
  ```

  Single precision, running on a Nehalem processor
  
  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;
  ```
  b[1] = b[0] + 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++)

\[
\text{Un}[i][j][k] = c \times (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] + U[i][j+1][k] + U[i][j][k-1] + U[i][j][k+1] - c2*b[i-1][j-1][k-1]);
\]

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
Alignment

- Unaligned data movement is expensive
- Accesses aligned on 16 byte boundaries go faster
- Intel compiler can “peel” and perform other optimizations

```c
double a[N], b[N];
for (int i = 1; i < N-1; i++)
a[i+1] = b[i] * 3;

char *x = ..
for (int i = 0; i < 1024; i++)
x[i] = 1;
```

```c
for (int i = 2; i < N-1; i++)
a[i+1] = b[i] * 3

peel = x & 0x0f;
if (peel != 0) {
    peel = 16 - peel;
    for (i = 0; i < peel; i++) x[i] = 1;
}
/* aligned access */
for (i = peel; i < 1024; i++) x[i] = 1;
```
Computing with Graphical Processing Units (GPUs)
Technological disruption

• New capabilities
• Changes the common wisdom for solving a problem including the implementation
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 (SM)
- 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>>>

David Kirk/NVIDIA & Wen-mei Hwu/UIUC
Coding example – Increment Array

• Rob Farber, Dr Dobb’s Journal

Serial Code

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

```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
2D thread blocks

```c
void addMatrix
    [float *a, float *b, float *c, int N]
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}

void main()
{
    ...
    addMatrix(a, b, c, N);
}
```

```c
__global__ void addMatrixG
    [float *a, float *b, float *c, int N]
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N(dimBlock.x, N,dimBlock.y);
    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

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
• 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
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);

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);
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

SM multithreaded Warp scheduler

warp 8 instruction 11
warp 1 instruction 42
warp 3 instruction 95
warp 8 instruction 12
warp 3 instruction 96

Courtesy David Kirk/NVIDIA
Wen-mei Hwu/UIUC