GPUs to the left
GPUs to the right
GPUs all day
GPUs all night

Chris Rossbach

cs378 Fall 2018
10/24/2018
Outline for Today

- Questions?
- Administrivia
  - Impending (minor) schedule changes
    - FPGA readings
    - Moved FPGA Lab Due Date
    - Barnes-Hut status change
  - Exam next week
- Agenda
  - CUDA
  - CUDA Performance
  - GPU parallel algorithms redux redux

Acknowledgements:
- http://www.seas.upenn.edu/~cis565/LECTURES/CUDA%20Tricks.pptx
Schedule Stuff

• Midterm Quiz questions posted
  • *No sheet of notes*
  • Monday: FGPAs or review?

• FPGA readings posted soon

• Lab 5 (FGPAs) due date moved

• Labs 6 (Barnes-Hut) “optional”
  • more time for project
  • You can do it as extra credit
Faux Quiz Questions

• How is occupancy defined (in CUDA nomenclature)?
• What’s the difference between a block scheduler (e.g. Giga-Thread Engine) and a warp scheduler?
• Modern CUDA supports UVM to eliminate the need for cudaMemcpy and cudaMemcpy*. Under what conditions might you want to use or not use it and why?
• What is control flow divergence? How does it impact performance?
• What is a bank conflict?
• What is work efficiency?
• What is the difference between a thread block scheduler and a warp scheduler?
• How are atomics implemented in modern GPU hardware?
• How is __shared__ memory implemented by modern GPU hardware?
• Why is __shared__ memory necessary if GPUs have an L1 cache? When will an L1 cache provide all the benefit of __shared__ memory and when will it not?
• Is cudaMemcpy still necessary after copyback if I have just one CUDA stream?
Ummm....
Review: Blocks and Threads

- Most kernels use both blockIdx.x and threadIdx.x
- Index an array with one elem. per thread (8 threads/block)

```c
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

What if my array size N % M != 0 😢

```c
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Why have threads?
- Why not just blocks or just threads?
- Unlike parallel blocks, threads can:
  - Communicate
  - Synchronize

- With M threads/block, unique index per thread is:

```c
int index = threadIdx.x + blockIdx.x * M;
```
Motivation: Stencils

• Each pixel → function of neighbors

• Edge detection:

\[
G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \text{ and } G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \]

• Blur:

<table>
<thead>
<tr>
<th>1/16</th>
<th>1/8</th>
<th>1/16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/8</td>
<td>1/4</td>
<td>1/8</td>
</tr>
<tr>
<td>1/16</td>
<td>1/8</td>
<td>1/16</td>
</tr>
</tbody>
</table>
1D Stencil

• Consider 1D stencil over 1D array of elements
  • Each output element is the sum of input elements within a radius

• Radius == 3 → each output element is sum of 7 input elements:
Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block

- Input elements read many times
  - With radius 3, each input element is read seven times

```c
__global__ void stencil_1d(int *in, int *out) {
    // note: idx comp & edge conditions omitted...
    int result = 0;
    for (int offset = -R; offset <= R; offset++)
        result += in[idx + offset];

    // Store the result
    out[idx] = result;
}
```
Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven times

```c
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```

Why is this a problem?
Sharing Data Between Threads

• Terminology: within a block, threads share data via *shared memory*

• Extremely fast on-chip memory, user-managed

• Declare using `__shared__`, allocated per block

• Data is *not visible* to threads in other blocks
Stencil with Shared Memory

• Cache data in shared memory
  – Read (blockDim.x + 2 * radius) elements from memory to shared
  – Compute blockDim.x output elements
  – Write blockDim.x output elements to global memory

  – Each block needs a **halo** of radius elements at each boundary
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
Data Race!

- The stencil example will not work...

- Suppose thread 15 reads the halo before thread 0 has fetched it...

```c
temp[lindex] = in[gindex]; Store at temp[18]
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS]; Skipped, threadIdx > RADIUS
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1]; Load from temp[19]
```
__syncthreads()

- void __syncthreads();

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block
Correct Stencil Kernel

Why doesn’t L1 cache provide same benefit?

- Manual control avoids eviction
- Write-through overheads avoided
- No write-back of dead values
- Provides an opportunity to make mis-aligned / bank conflicting accesses aligned/non-conflicting
Notes on __syncthreads()

• **void __syncthreads();**

• Synchronizes all threads within a block
  – Used to prevent RAW / WAR / WAW hazards

• All threads must reach the barrier
  – In conditional code, the condition must be uniform across the block

```c
__global__ void some_kernel(int *in, int *out) {
    // good idea?
    if(threadIdx.x == SOME_VALUE)
        __syncthreads();
}

__device__ void lock_trick(int *in, int *out) {
    __syncthreads();
    if(myIndex == 0)
        critical_section();
    __syncthreads();
}
```
GPU Atomics

Race conditions –
• Traditional locks are to be avoided
• How do we synchronize?

Read-Modify-Write – atomic

Is this a good idea?
Recap

• Launching parallel threads
  • Launch $N$ blocks with $M$ threads per block with `kernel<<<N,M>>>(...)`
  • Use `blockIdx.x` to access block index within grid
  • Use `threadIdx.x` to access thread index within block

• Allocate elements to threads:
  
  ```
  int index = threadIdx.x + blockIdx.x * blockDim.x
  ```

Use **__shared__** to declare a variable/array in shared memory
  
  Data is shared between threads in a block
  Not visible to threads in other blocks

Use **__syncthreads()** as a barrier
  
  Use to prevent data hazards

Atomics: watch out for control divergence
MANAGING THE DEVICE

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Coordinating Host & Device

• Kernel launches are **asynchronous**
  • Control returns to the CPU immediately

• CPU needs to synchronize before consuming the results

`cudaMemcopy()`  
Blocks the CPU until the copy is complete  
Copy begins when all preceding CUDA calls have completed

`cudaMemcopyAsync()`  
Asynchronous, does not block the CPU

`cudaDeviceSynchronize()`  
Blocks the CPU until all preceding CUDA calls have completed
Reporting Errors

• All CUDA API calls return an error code (cudaError_t)
  • Error in the API call itself
  OR
  • Error in an earlier asynchronous operation (e.g. kernel)

• Get the error code for the last error:
  cudaError_t cudaGetLastError(void)

• Get a string to describe the error:
  char *cudaGetErrorString(cudaError_t)

  printf("%s\n", cudaGetErrorString(cudaGetLastError()));
Device Management

• Application can query and select GPUs
  
  \texttt{cudaGetDeviceCount}(int \ *count)  
  \texttt{cudaSetDevice}(int \ device)  
  \texttt{cudaGetDevice}(int \ *device)  
  \texttt{cudaGetDeviceProperties}(cudaDeviceProp \ *prop, \ int \ device)  

• Multiple threads can share a device

• A single thread can manage multiple devices
  
  \texttt{cudaSetDevice} (i) \ to \ select \ current \ device  
  \texttt{cudaMemcpy} (...) \ for \ peer-to-peer \ copies$^\text{†}$

\textsuperscript{†} requires OS and device support
CUDA Events: Measuring Performance

```c
float memsettime;
cudaEvent_t start, stop;

// initialize CUDA timer
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start, 0);

// CUDA Kernel
...

// stop CUDA timer
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&memsettime, start, stop);
printf(" *** CUDA execution time: %f *** \n", memsettime);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```
GPU Memory Hierarchy

- SM-0
  - Registers
  - C
  - L1&SMEM
  - TEX

- SM-1
  - Registers
  - C
  - L1&SMEM
  - TEX

- SM-N
  - Registers
  - C
  - L1&SMEM
  - TEX

- L2

- Global Memory
Constant Cache

- Global variables marked by `__constant__`
  - `constant` and can’t be changed in device.
- Will be cached by Constant Cache
- Located in global memory
- Good for threads that access the same address

```c
__constant__ int a=10;
__global__ void kernel()
{
    a++; //error
}
```

Memory addresses
Texture Cache

- **Save Data as Texture:**
  - Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
  - Read-only data cache holds fetched samples
  - Backed up by the L2 cache

- **Why use it?**
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
  - What if your problem takes a large number of read-only points as input?
How many threads/blocks should I use?

// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;

• Usually things are correct if grid*block dims >= input size
• Getting good performance is another matter
Internals

__host__

void vecAdd()
{
    dim3 DimGrid = (ceil(n/256,1,1));
    dim3 DimBlock = (256,1,1);
    addKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}

__global__

void addKernel(float *A_d,
               float *B_d,
               float *C_d,
               int n){
    int i = blockIdx.x * blockDim.x
            + threadIdx.x;
    if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
Kernel Launch

- Commands by host issued through **streams**
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently

- Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)
  - Multiple streams mapped to each queue → serializes some kernels

- Kernel launch distributes thread blocks to SMs
Thread Blocks, Warps, Scheduling

- What’s a warp?
- Suppose one TB (threadblock) has 64 threads (2 warps)

- SMs split blocks into warps
- Unit of HW scheduling for SM
- 32 threads each
TBs, Warps, & Utilization

• 1 TB $\rightarrow$ 64 threads or 2 warps
TBs, Warps, & Utilization

- 1 TB $\rightarrow$ 64 threads or 2 warps
TBs, Warps, & Utilization

- 1 TB → 64 threads or 2 warps

Remaining TBs are queued
SIMD vs. SIMT

**Flynn Taxonomy**

- **SISD**
  - Single Instruction, Single Data
  - Loosely synchronized threads
  - e.g., pthreads

- **SIMD**
  - Single Instruction, Multiple Data
  - Synchronous operation
  - e.g., SSE/AVX

- **MISD**
  - Multiple Instruction, Single Data

- **MIMD**
  - Multiple Instruction, Multiple Data
  - Multiple threads
  - e.g., PTX, HSA

**Register File**

Single Scalar Thread

Loosely synchronized threads

Multiple threads
A Taco Bar

• Where is the parallelism here?
GPU: a multi-lane Taco Bar

- Where is the parallelism here?
GPU: a multi-lane Taco Bar

- Where is the parallelism here?
- There's none!
- This only works if you can keep every lane full at every step
- Throughput == Performance
- Goal: *Increase Occupancy*!
Where is the parallelism here?

• There’s none!
• This only works if you can keep every lane full at every step
• Throughput == Performance
• Goal: Increase Occupancy!
GPU Performance Metric: Occupancy

- **Occupancy** = (#Active Warps) / (#MaximumActive Warps)
  - Measures how well concurrency/parallelism is utilized
- Occupancy captures
  - *which resources* can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enables device-specific online tuning of kernel parameters

Shouldn’t we just create as many threads as possible?
Hardware Resources Are Finite

- SM – Stream Multiprocessor
- SP – Stream Processor

- Thread Block Control
  - Limits the #thread blocks

- Warp Schedulers
  - Limits the #threads

- Warp Context
  - Limits the #threads

- Register File
  - Limits the #threads

- L1/Shared Memory
  - Limits the #thread blocks
CUDA Occupancy

- Occupancy = (#Active Warps) / (#MaximumActive Warps)
  - Measure of how well max capacity is utilized

- Limits on the numerator:
  - Registers/thread
  - Shared memory/thread block
  - Number of scheduling slots: blocks, warps

- Limits on the denominator:
  - Memory bandwidth
  - Scheduler slots

What is the performance impact of varying kernel resource demands?
Impact of Thread Block Size

• Consider Fermi: 1536 threads/SM
  • With 512 threads/block how many blocks can execute (per SM) at an instant?
  • With 128 threads/block?

• Consider HW limit of 8 thread blocks/SM @ 128 threads/block:
  • Suppose only 1024 active threads at a time
  • Occupancy = 0.666 (1024/1536)

• To maximize utilization, thread block size should balance
  • demand for thread blocks vs.
  • thread slots
Impact of #Registers Per Thread

• Assume 10 registers/thread and a thread block size of 256
• Number of registers per SM = 16K
• A TB requires 2560 registers for a maximum of 6 thread blocks per SM
  • Uses all 1536 thread slots (6 blocks * 256 threads/block)
  • $2560 \text{ regs/block} \times 6 \text{ block/SM} = 15360 \text{ registers}$
• What is the impact of increasing number of registers by 2?
  • Granularity of management is a thread block!
  • Loss of concurrency of 256 threads!
  • $(12 \text{ regs/thread} \times 256 \text{ threads/block} \times 5 \text{ blocks/SM} = 15360 \text{ registers})$
Impact of Shared Memory

• Shared memory is allocated per thread block
  • Can limit the number of thread blocks executing concurrently per SM
• gridDim and blockDim parameters impact demand for
  • shared memory
  • number of thread slots
  • number of thread block slots
Balance

- **Navigate the tradeoffs**
  - maximize core utilization and memory bandwidth utilization
  - Device-specific
- **Goal**: Increase occupancy until one or the other is saturated

```c
template < class T >
__host__ cudaError_t cudaOccupancyMaxActiveBlocksPerMultiprocessor ( int* numBlocks, T func, int blockSize, size_t dynamicSMemSize ) [inline]
```

Returns occupancy for a device function.

**Parameters**
- **numBlocks**
  - Returned occupancy
- **func**
  - Kernel function for which occupancy is calculated
- **blockSize**
  - Block size the kernel is intended to be launched with
- **dynamicSMemSize**
  - Per-block dynamic shared memory usage intended, in bytes
Parallel Memory Accesses

• **Coalesced** main memory access (16/32x faster)
  • HW combines multiple warp memory accesses into a single coalesced access

• **Bank-conflict-free** shared memory access (16/32)
  • No alignment or contiguity requirements
    • CC 1.3: 16 different banks per half warp or same word
    • CC 2.x+3.0 : 32 different banks + 1-word broadcast each
Parallel Memory Architecture

• In a parallel machine, many threads access memory
  • Therefore, memory is divided into banks
  • Essential to achieve high bandwidth

• Each bank can service one address per cycle
  • A memory can service as many simultaneous accesses as it has banks

• Multiple simultaneous accesses to a bank result in a bank conflict
  • Conflicting accesses are serialized
Coalesced Main Memory Accesses

**single coalesced access**

**one and two coalesced accesses***
Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing
    - stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation
Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing
  - stride == 2

- 8-way Bank Conflicts
  - Linear addressing
  - stride == 8
Linear Addressing

• Given:

```c
__shared__ float shared[256];
float foo =
shared[baseIndex + s * threadIdx.x];
```

• This is only bank-conflict-free if s shares no common factors with the number of banks
  • 16 on G80, so s must be odd
Layered abstractions

* 1:1 correspondence between OS-level and user-level abstractions
* Diverse HW support enabled HAL
GPU abstractions

1. No kernel-facing API
2. OS resource-management limited
3. Poor composability
No OS support $\rightarrow$ No isolation

- Image convolution in CUDA
- Windows 7 x64 8GB RAM
- Intel Core 2 Quad 2.66GHz
- nVidia GeForce GT230

CPU+GPU schedulers not integrated!
...other pathologies abundant
Composition: Gestural Interface

- Raw images
- "Hand" events
- Geometric transformation
- Noise filtering
- Requires OS mediation
- High data rates
- Abundant data parallelism
  ...use GPUs!

10/24/2018
What We’d Like To Do

#> capture | xform | filter | detect &

- Modular design
  - flexibility, reuse
- Utilize heterogeneous hardware
  - Data-parallel components ➔ GPU
  - Sequential components ➔ CPU
- Using OS provided tools
  - processes, pipes
GPU Execution model

- GPUs cannot run OS:
  - different ISA
  - Memories have different coherence guarantees
    - (disjoint, or require fence instructions)

- Host CPU must “manage” GPU execution
  - Program inputs explicitly transferred/bound at runtime
  - Device buffers pre-allocated
Data migration

capture | xform | filter | detect &

#> capture | xform | filter | detect &

user

read() write() read() write() read() write() read()
capture xform filter detect

kernel

camdrv GPU driver HIDdrv

copy to GPU copy to GPU copy to GPU IRP

copy | xform | filter | detect

HW

GPU Run!

copy/xfer copy/xfer copy/xfer

10/24/2018
Device-centric APIs considered harmful

Matrix
gemm(Matrix A, Matrix B) {
  copyToGPU(A);
  copyToGPU(B);
  invokeGPU();
  Matrix C = new Matrix();
  copyFromGPU(C);
  return C;
}

What happens if I want the following?
Matrix D = A \times B \times C
Composed matrix multiplication

Matrix

AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(A,B);
    Matrix AxBxC = gemm(AxB,C);
    return AxBxC;
}

Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
Composed matrix multiplication

Matrix

AXBxXC(Matrix A, B, C) {
    Matrix AxB = gemm(A, B);
    Matrix AxBxC = gemm(AxB, C);
    return AxBxC;
}

Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
Composed matrix multiplication

Matrix

`AXBxC(Matrix A, B, C) {`  
  `Matrix AxB = gemm(A,B);`  
  `Matrix AXBxC = gemm(AXB, C);`  
  `return AXBxC;`  
`}`

...only to be copied right back!
What if I have many GPUs?

```java
Matrix
gemm(Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}
```
What if I have many GPUs?

Matrix gemm(CPU dev, Matrix A, Matrix B) {
    copyToGPU(dev, A);
    copyToGPU(dev, B);
    invokeGPU(dev);
    Matrix C = new Matrix();
    copyFromGPU(dev, C);
    return C;
}

What happens if I want the following?
Matrix D = A x B x C
Composition with many GPUs

```java
Matrix gemm(GPU dev, Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}

Matrix AxBxC(Matrix A, B, C) {
    Matrix AxB = gemm(???, A, B);
    Matrix AxBxC = gemm(???, AxB, C);
    return AxBxC;
}
```
Composition with many GPUs

Matrix gemm(GPU dev, Matrix A, Matrix B) {
    copyToGPU(A);
    copyToGPU(B);
    invokeGPU();
    Matrix C = new Matrix();
    copyFromGPU(C);
    return C;
}

Matrix AxBxC(GPU dev, Matrix A,B,C) {
    Matrix AxB = gemm(dev, A,B);
    Matrix AxBxC = gemm(dev, AxB,C);
    return AxBxC;
}

Rats...now I can only use 1 GPU. How to partition computation?
Composition with many GPUs

This will never be manageable for many GPUs. Programmer implements scheduling using static view!

Matrix
gemm(GPU dev, Matrix A, Matrix B)
{
copyToGPU(A);
copyToGPU(B);
invokeGPU();
Matrix C = new Matrix();
copyFromGPU(C);
return C;
}

Matrix
AxBxC(GPU devA, GPU devB, Matrix A,B,C) {
    Matrix AxB = gemm(devA, A,B);
    Matrix AxBxC = gemm(devB, AxB,C);
    return AxBxC;
}

Why don’t we have this problem with CPUs?
Dataflow: a better abstraction

- Nodes → computation
- Edges → communication
- Expresses parallelism explicitly
- Minimal specification of data movement: runtime does it.
- Asynchrony is a runtime concern (not programmer concern)
- No specification of compute → device mapping: like threads!
Advanced topics: Prefix-Sum

• in: 3 1 7 0 4 1 6 3
• out: 0 3 4 11 11 14 16 22
Trivial Sequential Implementation

void scan(int* in, int* out, int n)
{
    out[0] = 0;
    for (int i = 1; i < n; i++)
        out[i] = in[i-1] + out[i-1];
}
Parallel Scan

\[
\text{for}(d = 1; \ d < \log_2 n; \ d++)
\]

\[
\text{for all } k \text{ in parallel}
\]

\[
\text{if}( \ k \geq 2^d)
\]

\[
x[\text{out}][k] = x[\text{in}][k - 2^{d-1}] + x[\text{in}][k]
\]

\[
\text{else}
\]

\[
x[\text{out}][k] = x[\text{in}][k]
\]

Complexity $O(n\log_2 n)$
A work efficient parallel scan

• Goal is a parallel scan that is $O(n)$ instead of $O(n\log_2 n)$

• Solution:
  • Balanced Trees: Build a binary tree, sweep it to and from the root.
  • Binary tree with $n$ leaves has
    • $d=\log_2 n$ levels,
    • each level $d$ has $2^d$ nodes
  * One add is performed per node $\rightarrow O(n)$ add on a single traversal of the tree.
O(n) unsegmented scan

• Reduce/Up-Sweep
  for (d = 0; d < \log_2{n-1}; d++)
    for all k=0; k < n-1; k+=2^{d+1} in parallel
      x[k+2^{d+1}-1] = x[k+2^d-1] + x[k+2^{d+1}-1]

• Down-Sweep
  x[n-1] = 0;
  for (d = \log_2{n - 1}; d >=0; d--)
    for all k = 0; k < n-1; k += 2^{d+1} in parallel
      t = x[k + 2^d - 1]
      x[k + 2^d - 1] = x[k + 2^{d+1} -1]
      x[k + 2^{d+1} - 1] = t + x[k + 2^{d+1} - 1]
Tree analogy
O(n) Segmented Scan

Up-Sweep

1: for $d = 1$ to $\log_2 n - 1$ do
2: for all $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
3: if $f[k + 2^{d+1} - 1]$ is not set then
4: $x[k + 2^{d+1} - 1] \leftarrow x[k + 2^d - 1] + x[k + 2^d+1 - 1]$
5: $f[k + 2^{d+1} - 1] \leftarrow f[k + 2^d - 1] \mid f[k + 2^d+1 - 1]$
• Down-Sweep

1: \( x[n-1] \leftarrow 0 \)
2: for \( d = \log_2 n - 1 \) down to 0 do
3: for all \( k = 0 \) to \( n - 1 \) by \( 2^{d+1} \) in parallel do
4: \( t \leftarrow x[k + 2^d - 1] \)
5: \( x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1] \)
6: if \( f[k + 2^d] \) is set then
7: \( x[k + 2^{d+1} - 1] \leftarrow 0 \)
8: else if \( f[k + 2^d - 1] \) is set then
9: \( x[k + 2^{d+1} - 1] \leftarrow t \)
10: else
11: \( x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1] \)
12: Unset flag \( f[k + 2^d - 1] \)
Features of segmented scan

• 3 times slower than unsegmented scan
• Useful for building broad variety of applications which are not possible with unsegmented scan.
Primitives built on scan

• Enumerate
  • enumerate([t f f t f t]) = [0 1 1 2 2 3]
  • Exclusive scan of input vector

• Distribute (copy)
  • distribute([a b c][d e]) = [a a a][d d]
  • Inclusive scan of input vector

• Split and split-and-segment

  Split divides the input vector into two pieces, with all the elements marked false on the left side of the output vector and all the elements marked true on the right.
Applications

• Quicksort
• Sparse Matrix-Vector Multiply
• Tridiagonal Matrix Solvers and Fluid Simulation
• Radix Sort
• Stream Compaction
• Summed-Area Tables
Quicksort

[5 3 7 4 6]  # initial input
[5 5 5 5 5]  # distribute pivot across segment
[f f t f t]  # input > pivot?
[5 3 4][7 6]  # split-and-segment
[5 5 5][7 7]  # distribute pivot across segment
[t f f][t f]  # input >= pivot?
[3 4 5][6 7]  # split-and-segment, done!
Sparse Matrix-Vector Multiplication

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2
\end{bmatrix}
+ \begin{bmatrix}
a & 0 & b \\
c & d & e \\
0 & 0 & f
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2
\end{bmatrix}
\]

\[
\text{value} = [a, b, c, d, e, f] \\
\text{index} = [0, 2, 0, 1, 2, 2] \\
\text{rowPtr} = [0, 2, 5]
\]

\[
\text{product} = [x_0a, x_2b, x_0c, x_1d, x_2e, x_2f] \\
= [(x_0a, x_2b)[x_0c, x_1d, x_2e][x_2f]] \\
= [(x_0a + x_2b, x_2b)] \\
\quad [x_0c + x_1d + x_2e, x_1d + x_2e, x_2e][x_2f]] \\
y = y + [(x_0a + x_2b, x_0c + x_1d + x_2e, x_2f] \\
\]

1. The first kernel runs over all entries. For each entry, it sets the corresponding flag to 0 and performs a multiplication on each entry: \(\text{product} = x[\text{index}] \times \text{value}\).
2. The next kernel runs over all rows and sets the head flag to 1 for each rowPtr in flag through a scatter. This creates one segment per row.
3. We then perform a backward segmented inclusive sum scan on the \(e\) elements in product with head flags in flag.
4. To finish, we run our final kernel over all rows, adding the value in \(y\) to the gathered value from products [idx].
Stream Compaction

Definition:
- Extracts the ‘interest’ elements from an array of elements and places them continuously in a new array

Uses:
- Collision Detection
- Sparse Matrix Compression
Stream Compaction

Input: We want to preserve the gray elements
Set a ‘1’ in each gray input

Scan

Scatter gray inputs to output using scan result as scatter address
Radix Sort Using Scan

Input Array

b = least significant bit

e = Insert a 1 for all false sort keys

f = Scan the 1s

Total Falses = e[n-1] + f[n-1]

t = index – f + Total Falses

d = b ? t : f

Scatter input using d as scatter address
Specialized Libraries

• CUDPP: CUDA Data Parallel Primitives Library
  • CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum ("scan"), parallel sort and parallel reduction.
CUDPP_DLL CUDPPResult cudppSparseMatrixVectorMultiply(CUDPPHandle sparseMatrixHandle, void * d_y, const void * d_x)

Perform matrix-vector multiply \( y = A \times x \) for arbitrary sparse matrix \( A \) and vector \( x \).
CUDPPScanConfig config;
    config.direction = CUDPP_SCAN_FORWARD; config.exclusivity = CUDPP_SCAN_EXCLUSIVE; config.op = CUDPP_ADD;
    config.datatype = CUDPP_FLOAT; config.maxNumElements = numElements;
    config.maxNumRows = 1;
    config.rowPitch = 0;
cudppInitializeScan(&config);
cudppScan(d_odata, d_idata, numElements, &config);
CUFFT

• No. of elements<8192 slower than fftw
• >8192, 5x speedup over threaded fftw and 10x over serial fftw.
CUBLAS

• Cuda Based Linear Algebra Subroutines
• Saxpy, conjugate gradient, linear solvers.
• 3D reconstruction of planetary nebulae.
  • http://graphics.tu-bs.de/publications/Fernandez08TechReport.pdf