Outline for Today

• Questions?
• Administrivia
  • Exam graded soon!
• Agenda
  • CUDA

Acknowledgements:

• http://www.seas.upenn.edu/~cis565/LECTURES/CUDA%20Tricks.pptx
• http://ece757.ece.wisc.edu/lect13-gppgpu.pptx
Faux Quiz Questions (5 min, any 2)

1. Why do languages like CUDA and OpenCL have both blocks and threads?
2. What does the __shared__ keyword do? How is it implemented by HW?
3. CUDA kernels have implicit barrier synchronization: __syncthreads(). Is it necessary? Why or why not?
4. Is traditional locking implementable on a GPU? What alternatives are there for synchronizing / avoiding races?
5. How is occupancy defined (in CUDA nomenclature)?
6. What is control flow divergence? How does it impact performance?
7. What is a bank conflict?
8. What is the difference between a thread block scheduler and a warp scheduler?
10. In CUDA, what is the difference between grids, blocks, and threads? What is their counterpart in OpenCL?
Parallel Algorithms Redux

Key idea:

*Express sequential algorithms as combinations of parallel patterns*

Examples:

- Map
- Reductions
- Scans
- Re-orderings (scatter/gather/sort)
Example: Parallel GroupBy

- Group a collection by key
- Lambda function maps elements $\rightarrow$ key

```csharp
var res = ints.GroupBy(x => x);
```

```
foreach (T elem in PF(ints))
{
    key = KeyLambda(elem);
    group = GetGroup(key);
    group.Add(elem);
}
```

- Insufficient Parallelism
- Requires synchronization
Parallel GroupBy

Process each input element in parallel

- grouping ~ shuffling
- input item → output offset such that groups are contiguous
- output offset = group offset + item number
- … but how to get the group offset, item number?

```
ints       res
10 30 20 10 20 30 10
10 10 10 30 30 20 20
```

- Start index of each group in the output sequence
- Number of elements in each group
- Number of groups and input → group mapping
GroupBy using parallel primitives

Assign group IDs
Group ID:
10 20 30
0 1 2

Compute group sizes
Group ID:
10 20 30
0 1 2
Group Size:
3 2 2

Compute start indices
Group ID:
10 20 30
0 1 2
Group Start Index:
0 3 5

Write Outputs
10 10 10 20 20 30 30

Sorting or hashing
Hash table lookup: group ID
-- Uses atomic increment
-- map

prefix sum of group sizes
Write to output location
– Uses atomic increment
– Scatter gather

We’ll revisit after more CUDA background…
Each SM has multiple vector units (4)
  • 32 lanes wide $\rightarrow$ warp size
  • Vector units use **hardware multi-threading**
  • Execution $\rightarrow$ a grid of thread blocks (TBs)
    • Each TB has some number of threads
What is CUDA?

• CUDA Architecture
  • Expose GPU parallelism for general-purpose computing
  • Retain performance

• CUDA C/C++
  • Based on industry-standard C/C++
  • Small set of extensions to enable heterogeneous programming
  • Straightforward APIs to manage devices, memory etc.
CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
  - __syncthreads() 
- Asynchronous operation
- Handling errors
- Managing devices
HELLO WORLD!

CONCEPTS
- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Heterogeneous Computing

- **Terminology:**
  - *Host*  The CPU and its memory (host memory)
  - *Device*  The GPU and its memory (device memory)
Heterogeneous Computing

Parallel fn

Serial code

Parallel code

Serial code
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
int main(void) {
    printf("Hello World!\n");
    return 0;
}

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Output:

$ nvcc hello_world.cu
$ a.out
$ Hello World!
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<1,1>>();
    printf("Hello World!\n");
    return 0;
}
```

- Two new syntactic elements...
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`
Hello World! with Device COde

mykernel<<<1,1>>>();

• Triple angle brackets mark a call from host code to device code
  • Also called a “kernel launch”
  • We’ll return to the parameters (1,1) in a moment

• That’s all that is required to execute a function on the GPU!
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

• `mykernel()` does nothing, somewhat anticlimactic!

Output:

```
$ nvcc hello.cu
hello.cu
$ a.out
Hello World!
$ 
```
Parallel Programming in CUDA C/C++

• But wait... GPU computing is about massive parallelism!

• We need a more interesting example...

• We’ll start by adding two integers and build up to vector addition
Addition on the Device

• A simple kernel to add two integers

```c
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

• As before `__global__` is a CUDA C/C++ keyword meaning
  • `add()` will execute on the device
  • `add()` will be called from the host
Addition on the Device

• Note that we use pointers for the variables

```c
#include <cuda_runtime.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

• `add()` runs on the device, so `a`, `b` and `c` must point to device memory

• We need to allocate memory on the GPU
Memory Management

• Host and device memory are separate entities
  • Device pointers point to GPU memory
    May be passed to/from host code
    May not be dereferenced in host code
  • Host pointers point to CPU memory
    May be passed to/from device code
    May not be dereferenced in device code

• Simple CUDA API for handling device memory
  • cudaMalloc(), cudaFree(), cudaMemcpy()
  • Similar to the C equivalents malloc(), free(), memcpy()
Addition on the Device: \texttt{add()} \\

• Returning to our \texttt{add()} kernel \\

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
\end{verbatim}

• Let’s take a look at main()...
Addition on the Device: `main()`

```c
int main(void) {
    int a, b, c;             // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;
}
```
Addition on the Device: `main`

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

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

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

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
```
RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Getting Parallel

• GPU computing is about massive parallelism
  • So how do we run code in parallel on the device?

\[
\text{add}<<<1, 1 >>>();
\]

\[
\text{add}<<<N, 1 >>>();
\]

• Instead of executing \texttt{add()} once, execute \(N\) times in parallel
Vector Addition on the Device

• With `add()` running in parallel we can do vector addition

• Terminology: each parallel invocation of `add()` is a block
  • The set of blocks is referred to as a grid
  • Each invocation can refer to its block index using `blockIdx.x`

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

• By using `blockIdx.x` to index into the array, each block handles a different index
Vector Addition on the Device

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:

```
Block 0
  c[0] = a[0] + b[0];

Block 1
  c[1] = a[1] + b[1];

Block 2

Block 3
```
Vector Addition on the Device: \texttt{add()}  

• Returning to our parallelized \texttt{add()} kernel

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
\end{verbatim}

• Let’s take a look at \texttt{main()}...
Vector Addition on the Device: main()

```c
#define N 512
int main(void) {
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
}
```
Vector Addition on the Device: `main()`

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

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(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;
```
INTRODUCING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
CUDA Threads

• Terminology: a block can be split into parallel threads

• Change `add()` to use parallel *threads* instead of parallel *blocks*:

```c
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

• Use `threadIdx.x` instead of `blockIdx.x`

• Need to make one change in `main()`...
Vector Addition Using Threads: `main()`

```c
#define N 512
int main(void) {
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
}
```
Vector Addition Using Threads: main()

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

   // Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

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

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

<table>
<thead>
<tr>
<th></th>
<th>Traditional CPU</th>
<th>Graphics Shaders</th>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD lane</td>
<td>thread</td>
<td>thread</td>
<td>thread</td>
<td>work-item</td>
</tr>
<tr>
<td>~thread</td>
<td>-</td>
<td>-</td>
<td>warp</td>
<td>-</td>
</tr>
<tr>
<td>thread group</td>
<td>block</td>
<td>block</td>
<td>work group</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>grid</td>
<td>grid</td>
<td>N-D range</td>
<td></td>
</tr>
</tbody>
</table>
COMBINING THREADS AND BLOCKS
Combining Blocks and Threads

• We’ve seen parallel vector addition using:
  • Many blocks with one thread each (M:1)
  • One block with many threads (1:M)

• How to make vector addition to use both blocks and threads?

• How to deal with blockIdx.* vs threadIdx.*?
Indexing Arrays with Blocks and Threads

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

  ```
  int index = threadIdx.x + blockIdx.x * M;
  ```

• With M threads/block, unique index per thread is:
Indexing Arrays: Example

- Which thread will operate on the red element?
  - M=8 Threads, 4 blocks

\[
\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} \times M;
\]
\[
= 5 + 2 \times 8;
\]
\[
= 21;
\]
Vector Addition with Blocks and Threads

• Use the built-in variable `blockDim.x` for threads per block

```c
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

• Combined `add()` using parallel threads and blocks

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

• What changes need to be made in `main()`?
Addition with Blocks and Threads:

main()

#define N (2048*2048)
#define THREADS_PER_BLOCK 512

int main(void) {

    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
}
Addition with Blocks and Threads:

main()

    // 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;

Anyone see a problem?
Handling Arbitrary Vector Sizes

• Typical problems are not friendly multiples of `blockDim.x`

• Avoid accessing beyond the end of the arrays:

```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];
}
```

• Update the kernel launch:

```c
add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
```
Why Bother with Threads?

• Threads seem unnecessary
  • They add a level of complexity
  • What do we gain?

• Unlike parallel blocks, threads have mechanisms to:
  • Communicate
  • Synchronize

• To look closer, we need a new example...
COOPERATING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
Stencils

• Each pixel → function of neighbors

• Edge detection:

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

• Blur:

\[
\begin{array}{ccc}
1/16 & 1/8 & 1/16 \\
1/8 & 1/4 & 1/8 \\
1/16 & 1/8 & 1/16 \\
\end{array}
\]
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

__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) {
  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 \((\text{blockDim}.x + 2 \times \text{radius})\) elements from memory to shared
  – Compute \text{blockDim}.x output elements
  – Write \text{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;
}

Are we done?
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];
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()
__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];
    }
    __syncthreads();

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

    // Store the result
    out[gindex] = result;
}
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();
}
```
Atomics

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

```c
__device__ double atomicAdd(double* address, double val) {
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull,
                        assumed,
                        __double_as_longlong(val + __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
```

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
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

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

`cudaMemcpyAsync()`  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

\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);
```
Compute Capability

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

The following presentations concentrate on Fermi devices

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>Selected Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>Tesla models</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BFLOAT16-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Tensor Cores</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Mixed Precision Warp-Matrix Functions (Warp matrix functions)</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Hardware-accelerated async (Asynchronous Data Access)</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Hardware-accelerated Split-arrival Warp Barrier (Asynchronous Barrier)</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>L2 Cache Residency Management (Device Memory L2 Access Management)</td>
</tr>
</tbody>
</table>

Note that the KB and K units used in the following table correspond to 1024 bytes (i.e., a Kib) and 1024 respectively.

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of resident threads per grid (Concurrent Kernel Execution)</td>
<td>32</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>32</td>
</tr>
<tr>
<td>Maximum x-dimension of a grid of thread blocks</td>
<td>2^31-1</td>
</tr>
<tr>
<td>Maximum y- or z-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimensionality of a thread block</td>
<td>3</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>1024</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
</tbody>
</table>

| Maximum number of resident blocks per grid (Concurrent Kernel Execution) | 16 |
| Maximum number of resident warp threads per block (Concurrent Kernel Execution) | 64 |
GPU Memory Hierarchy
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
  • Read-only data cache
  • 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
__host__
Void vecAdd()
{
    dim3 DimGrid = (ceil(n/256,1,1);
    dim3 DimBlock = (256,1,1);
    vecAddKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}

__global__
void vecAddKernel(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?

• Imagine one TB (threadblock) has 64 threads (2 warps)
  • On K20c GK110 GPU: 13 SMs, max 64 warps/SM, max 32 concurrent kernels
TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
  - On K20c GK110 GPU: 13 SMX, max 64 warps/SM, max 32 concurrent kernels
TBs, Warps, & Utilization

• One TB has 64 threads or 2 warps

• On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels
TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
  - On K20c GK110 GPU: 13 SMX, max 64 warps/SM, max 32 concurrent kernels
NVIDIA GK110 (Kepler)

- Thread Block Scheduler
- Multiple Warp Schedulers
- 64K 32-bit registers
- 192 cores – 6 clusters of 32 cores each

Image from http://mandetech.com/2012/05/20/nvidia-new-gpu-and-visualization/
Execution Sequencing in a Single SM

Cycle level view: GPU kernel execution
Example: VectorAdd on GPU

**CUDA:**

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

**PTX (Assembly):**

```
setp.lt.s32 %p, %r5, %rd4; // r5 = index, rd4 = N
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6]; // r6 = &a[index]
ld.global.f32 %f2, [%r7]; // r7 = &b[index]
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3; // r8 = &c[index]

L2:
ret;
```
Example: VectorAdd on GPU

- N=8, 8 Threads, 1 block, warp size = 4
- 1 SM, 4 Cores
- Pipeline:
  - Fetch:
    - One instruction from each warp
    - Round-robin through all warps
  - Execution:
    - In-order execution within warps
    - With proper data forwarding
    - 1 Cycle each stage
- How many warps?
Execution Sequence

```assembly
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;

Execution Sequence (cont.)
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;

Execution Sequence (cont.)

setp
W0

setp
W1

setp
W0

setp
W1

setp
W0

setp
W1

setp
W0

setp
W1

setp
W0

setp
W1

setp
W0

setp
W1

setp
W0

setp
W1

setp
W0

setp
W1

setp
W0

Warp0

Warp1
Execution Sequence (cont.)

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;

Execution Sequence (cont.)
Execution Sequence (cont.)

```assembly
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
```

**Execution Sequence (cont.)**

```
ld    %f1, [W0]
bra   W1
bra   W0
     WB

ld    %f2, [W0]
bra   W1
bra   W0
     WB
```

```
Warp0  Warp1
```
Execution Sequence (cont.)

<table>
<thead>
<tr>
<th>Id W1</th>
<th>FE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id W0</td>
<td>DE</td>
</tr>
<tr>
<td></td>
<td>EXE</td>
</tr>
<tr>
<td></td>
<td>MEM</td>
</tr>
<tr>
<td></td>
<td>bra W1</td>
</tr>
<tr>
<td></td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;

\[
\]
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
L1:
l.load.global.f32 %f1, [%r6];
l.load.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;

Execution Sequence (cont.)
Execution Sequence (cont.)

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
```
Execution Sequence (cont.)

```
setp.lt.s32 %p, %r5, %rd4;
%p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;
```

Warp0

Warp1
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;
L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret;
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
Execution Sequence (cont.)

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;
L2:
ret
```

Execution Sequence (cont.)
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;

Execution Sequence (cont.)
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;

Execution Sequence (cont.)
setp.lt.s32 %p, %r5, %rd4; @p bra L1; bra L2;

L1:
  ld.global.f32 %f1, [%r6];
  ld.global.f32 %f2, [%r7];
  add.f32 %f3, %f1, %f2;
  st.global.f32 [%r8], %f3;

L2:
  ret;

Execution Sequence (cont.)
Execution Sequence (cont.)

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;
st.global.f32 [%r8], %f3;

L2:
ret;
```
Execution Sequence (cont.)

Note: Idealized execution without memory or special function delays
Resource Limits

Kernel Distributor

SM Scheduler

SM

SM

SM

SM

DRAM

Thread Block Control

TB 0

Warp Schedulers

Warp Context

Register File

L1/Shared Memory

Limits the #thread blocks

Limits the #threads

Limits the #threads

Limits the #thread blocks

SM – Stream Multiprocessor

SP – Stream Processor
Performance Metrics: *Occupancy*

- Performance implications of programming model properties
  - Warps
  - Thread blocks
  - Register file usage
- Occupancy captures
  - which resources can be dynamically shared
  - how to reason about resource demands of a CUDA kernel
  - Enable device-specific online tuning of kernel parameters
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: thread blocks or warps

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

What is the performance impact of varying kernel resource demands?
Programming Model Attributes

- #SMs
- Max Threads/Block
- Max Threads/Dim
- Warp size

To maximize occupancy

- Know micro-architecture of target processor
- Exercise kernel configuration parameters
Impact of Thread Block Size

• Consider Fermi with 1536 threads/SM
  • With 512 threads/block, can only up to 3 thread blocks executing at an instant
  • With 128 threads/block → 12 thread blocks per SM
  • Consider how many instructions can be in flight?

• Consider limit of 8 thread blocks/SM?
  • Only 1024 active threads at a time
  • Occupancy = 0.666

• 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
• What is the impact of increasing number of registers by 2?
  • Granularity of management is a thread block!
  • Loss of concurrency of 256 threads!
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
Parallel Memory Accesses

• **Coalesced** main memory access (16/32x faster)
  • Under some conditions, HW combines multiple (half) warp memory accesses into a single coalesced access
    • CC 1.3: 64-byte aligned 64-byte line (any permutation)
    • CC 2.x+3.0: 128-byte aligned 128-byte line (cached)

• **Bank-conflict-free** shared memory access (16/32)
  • No superword 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
Coalesced Main Memory Accesses

**Single Coalesced Access**
- Thread 0 accesses Address 128
- Thread 1 accesses Address 112
- Thread 2 accesses Address 116
- Thread 3 accesses Address 120
- Thread 4 accesses Address 124
- Thread 5 accesses Address 128
- Thread 6 accesses Address 132

**Two Coalesced Accesses**
- Thread 0 accesses Address 128
- Thread 1 accesses Address 132
- Thread 2 accesses Address 136
- Thread 3 accesses Address 140
- Thread 4 accesses Address 144
- Thread 5 accesses Address 148
- Thread 6 accesses Address 152

NVIDIA

CUDA Optimization Tutorial
Device Properties

```c
cudaError_t cudaGetDeviceProperties ( struct cudaDeviceProp * prop, int device )
```

Returns in *prop the properties of device *device*. The `cudaDeviceProp` structure is defined as:

```c
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int clockRate;
    size_t totalConstMem;
    int major;
    int minor;
    size_t textureAlignment;
    size_t texturePitchAlignment;
    int deviceOverlap;
    int multiProcessorCount;
}
```
CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong> int LocalVar;</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__

- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory
Memory Coalescing

- When accessing global memory, peak performance utilization occurs when all threads in a half warp access continuous memory locations.
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
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

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4

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

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL, University of Illinois, Urbana-Champaign
How addresses map to banks on G80

• Each bank has a bandwidth of 32 bits per clock cycle
• Successive 32-bit words are assigned to successive banks

• G80 has 16 banks
  • So bank = address % 16
  • Same as the size of a half-warps
    • No bank conflicts between different half-warps, only within a single half-warps
Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts

- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)

- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank
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
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

```c
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

for (d = 1; d < \log_2 n; d++)
    for all k in parallel
        if (k \geq 2^d)
            x[out][k] = x[in][k - 2^{d-1}] + x[in][k]
        else
            x[out][k] = x[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

\[
\text{for (d = 0; d < log}_2\text{n-1; d++)}
\]

\[
\text{for all k=0; k < n-1; k+=2}^{d+1}\text{ in parallel}
\]

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

• Down-Sweep

\[
x[n-1] = 0;
\]

\[
\text{for (d = log}_2\text{n - 1; d >=0; d--)}
\]

\[
\text{for all k = 0; k < n-1; k += 2}^{d+1}\text{ in parallel}
\]

\[
t = x[k + 2^d - 1]
\]

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

\[
x[k + 2^{d+1} - 1] = t + x[k + 2^{d+1} - 1]
\]
Tree analogy

\[
\sum(x_0, x_1) \quad \sum(x_4, x_5) \quad \sum(x_6, x_7)
\]

\[
\sum(x_0, x_2) \quad \sum(x_3, x_4) \quad \sum(x_5, x_6)
\]

\[
\sum(x_0, x_4) \quad \sum(x_1, x_5) \quad \sum(x_2, x_6)
\]

\[
\sum(x_0, x_3) \quad \sum(x_4, x_6)
\]
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: $f[k + 2^{d+1} - 1] \leftarrow f[k + 2^d - 1] | f[k + 2^{d+1} - 1]$
• 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

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

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

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

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: product = x[index] * 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**

<table>
<thead>
<tr>
<th>100</th>
<th>111</th>
<th>010</th>
<th>110</th>
<th>011</th>
<th>101</th>
<th>001</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>100</th>
<th>111</th>
<th>010</th>
<th>110</th>
<th>011</th>
<th>101</th>
<th>001</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>111</td>
<td>010</td>
<td>110</td>
<td>011</td>
<td>101</td>
<td>001</td>
<td>000</td>
</tr>
</tbody>
</table>

| 100 | 010 | 110 | 000 | 111 | 011 | 101 | 001 |

\[
e = \text{Insert a 1 for all false sort keys}
\]

\[
f = \text{Scan the 1s}
\]

\[
t = \text{index} - f + \text{Total Falses}
\]

\[
d = b ? t : f
\]

**Total Falses** = \(e[n-1] + f[n-1]\)

\[d = b \oplus t : f\]
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*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.