CS 677: Parallel Programming for Many-core Processors
Lecture 13

Instructor: Philippos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Filippos.Mordohai@stevens.edu
Outline

• Introduction to OpenMP
• Hardware developments
• Developments in CUDA
OpenMP

Based on tutorial by Joel Yliluoma
http://bisqwit.iki.fi/story/howto/openmp/
OpenMP in C++

• OpenMP consists of a set of compiler #pragmas that control how the program works.

• The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.
Simple Example

- Multiple threads

```c++
#include <cmath>
int main()
{
    const int size = 256;
    double sinTable[size];

    #pragma omp parallel for
    for(int n=0; n<size; ++n)
        sinTable[n] = std::sin(2 * M_PI * n / size);

    // the table is now initialized
}
```
Simple Example

- Single thread, SIMD

```cpp
#include <cmath>
int main()
{
    const int size = 256;
    double sinTable[size];

#pragma omp simd
for(int n=0; n<size; ++n)
    sinTable[n] = std::sin(2 * M_PI * n / size);

    // the table is now initialized
}
```
Simple Example

• Multiple threads on another device

```cpp
#include <cmath>
int main()
{
    const int size = 256;
    double sinTable[size];

    #pragma omp target teams distribute parallel for
        map(from:sinTable[0:256])
    for(int n=0; n<size; ++n)
        sinTable[n] = std::sin(2 * M_PI * n / size);

    // the table is now initialized
}
```
Syntax

• All OpenMP constructs start with `#pragma omp`

• The `parallel` construct
  – Creates a `team` of N threads (N determined at runtime) all of which execute statement or next block
  – All variables declared within block become local variables to each thread
  – Variables shared from the context are handled transparently, sometimes by passing a reference and sometimes by using register variables
if

extern int parallelism_enabled;
#pragma omp parallel for if(parallelism_enabled)
for(int c=0; c<n; ++c)
    handle(c);
#pragma omp for
for(int n=0; n<10; ++n)
{
    printf(" %d", n);
}
printf(".\n");

• Output may appear in arbitrary order
Creating a New Team

```c
#pragma omp parallel
{
    #pragma omp for
    for(int n=0; n<10; ++n) printf(" %d", n);
}
printf(".\n");
```

- Or, equivalently

```c
#pragma omp parallel for
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```
Specifying Number of Threads

```c
#pragma omp parallel num_threads(3)
{
    // This code will be executed by three threads.

    // Chunks of this loop will be divided amongst
    // the (three) threads of the current team.
    #pragma omp for
    for(int n=0; n<10; ++n) printf(" %d", n);
}
```
parallel, for, parallel for

The difference between parallel, parallel for and for is as follows:

- A team is the group of threads that execute currently.
  - At the program beginning, the team consists of a single thread.
  - A parallel construct splits the current thread into a new team of threads for the duration of the next block/statement, after which the team merges back into one.

- for divides the work of the for-loop among the threads of the current team. It does not create threads.

- parallel for is a shorthand for two commands at once. Parallel creates a new team, and for splits that team to handle different portions of the loop.

- If your program never contains a parallel construct, there is never more than one thread.
Scheduling

- Each thread independently decides which chunk of the loop it will process

```c
#pragma omp for schedule(static)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".
");
```

- In dynamic schedule, each thread asks OpenMP runtime library for an iteration number, then handles it and asks for next.
  - Useful when different iterations take different amounts of time to execute

```c
#pragma omp for schedule(dynamic)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".
");
```
Scheduling

• Each thread asks for iteration number, executes 3 iterations, then asks for another

```c
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(" .\n");
```
#pragma omp for ordered schedule(dynamic)
for(int n=0; n<100; ++n)
{
    files[n].compress();

    #pragma omp ordered
    send(files[n]);
}
reduction

```c
int sum=0;
#pragma omp parallel for reduction(+:sum)
for(int n=0; n<1000; ++n)
    sum += table[n];
```
Sections

#pragma omp parallel sections
{
    { Work1(); }  
#pragma omp section
    { Work2();
        Work3(); }
#pragma omp section
    { Work4(); }  
}

#pragma omp parallel // starts a new team
{
    //Work0(); // this function would be run by all threads.

    #pragma omp sections // divides the team into sections
    {
        // everything herein is run only once.
        { Work1(); }
    #pragma omp section
        { Work2();
            Work3(); }
    #pragma omp section
        { Work4(); }
    }

    //Work5(); // this function would be run by all threads.
SIMD means that multiple calculations will be performed simultaneously using special instructions that perform the same calculation to multiple values at once.

This is often more efficient than regular instructions that operate on single data values. This is also sometimes called vector parallelism or vector operations.

```cpp
float a[8], b[8];
...
#pragma omp simd
for(int n=0; n<8; ++n) a[n] += b[n];
```
```c
#pragma omp declare simd aligned(a,b:16)
void add_arrays(float *__restrict__ a, float *__restrict__ b)
{
    #pragma omp simd aligned(a,b:16)
    for(int n=0; n<8; ++n) a[n] += b[n];
}

Reduction:
int sum=0;
#pragma omp simd reduction(+:sum)
for(int n=0; n<1000; ++n) sum += table[n];
```
#pragma omp declare simd aligned(a,b:16)
    void add_arrays(float *__restrict__ a, float *__restrict__ b)
    {
        #pragma omp simd aligned(a,b:16)
        for(int n=0; n<8; ++n) a[n] += b[n];
    }

- Tells compiler that each element is aligned to the given number of bytes
- Increases performance
#pragma omp declare target

int x;
void murmur() { x+=5; }
#pragma omp end declare target

• This creates one or more versions of "x" and "murmur". A set that exists on the host computer, and also a separate set that exists and can be run on a device.
• These two functions and variables are separate, and may contain values separate from each others.
The target data construct creates a device data environment. The target construct executes the construct on a device (and also has target data features). These two constructs are identical in effect:

```c
#pragma omp target // device()... map()... if()...
{
    <<statements...>>
}
```

```c
#pragma omp target data // device()... map()... if()...
{
    #pragma omp target
    {
        <<statements...>>
    }
}
```

critical

• Restricts the execution of the associated statement / block to a single thread at a time.
• May optionally contain a global name that identifies the type of the critical construct. No two threads can execute a critical construct of the same name at the same time.

• Below, only one of the critical sections named "dataupdate" may be executed at any given time, and only one thread may be executing it at that time. I.e. the functions "reorganize" and "reorganize_again" cannot be invoked at the same time, and two calls to the function cannot be active at the same time.

```c
#pragma omp critical(dataupdate)
{
    datastructure.reorganize();
}
...
#pragma omp critical(dataupdate)
{
    datastructure.reorganize_again();
}
```
private, firstprivate, shared

int a, b=0;
#pragma omp parallel for private(a) shared(b)
for(a=0; a<50; ++a)
{
    #pragma omp atomic
    b += a;
}

private, firstprivate, shared

#include <string>
#include <iostream>

int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel private(a,c) shared(b)
        num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << ",
        b is (" << b << ")\n";
    }
}

• Outputs “k” not “xk”, c is uninitialized
private, firstprivate, shared

#include <string>
#include <iostream>

int main()
{
    std::string a = "x", b = "y";
    int c = 3;

    #pragma omp parallel firstprivate(a,c) shared(b)
    num_threads(2)
    {
        a += "k";
        c += 7;
        std::cout << "A becomes (" << a << ",", b is (" << b << ")\n";
    }
}

• Outputs “xk”
Barriers

#pragma omp parallel
{
    /* All threads execute this. */
    SomeCode();

#pragma omp barrier

    /* All threads execute this, but not before * all threads have finished executing
       SomeCode().
    */
    SomeMoreCode();
}
#pragma omp parallel
{
    #pragma omp for
    for(int n=0; n<10; ++n) Work();

    // This line is not reached before the for-loop is completely finished
    SomeMoreCode();
}

// This line is reached only after all threads from
// the previous parallel block are finished.
CodeContinues();

#pragma omp parallel
{
    #pragma omp for nowait
    for(int n=0; n<10; ++n) Work();

    // This line may be reached while some threads are still executing for-loop.
    SomeMoreCode();
}

// This line is reached only after all threads from
// the previous parallel block are finished.
CodeContinues();
Nested Loops

```c
#pragma omp parallel for
for(int y=0; y<25; ++y)
{
    #pragma omp parallel for
    for(int x=0; x<80; ++x)
    {
        tick(x,y);
    }
}
```

- Code above fails, inner loop runs in sequence

```c
#pragma omp parallel for collapse(2)
for(int y=0; y<25; ++y)
    for(int x=0; x<80; ++x)
    {
        tick(x,y);
    }
```
The Fermi Architecture

Selected notes from presentation by:
Michael C. Shebanow

Principal Research Scientist,
NV Research
mshebanow@nvidia.com

(2010)
Much Better Compute

- **Programmability**
  - C++ Support
  - Exceptions/Debug support

- **Performance**
  - Dual issue SMs
  - L1 cache
  - Larger Shared Memory
  - Much better DP math
  - Much better atomic support

- **Reliability**: ECC

<table>
<thead>
<tr>
<th></th>
<th>GT200</th>
<th>GF100</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Texture Cache (per quad)</td>
<td>12 KB</td>
<td>12 KB</td>
<td>Fast texture filtering</td>
</tr>
<tr>
<td>Dedicated L1 LD/ST Cache</td>
<td>X</td>
<td>16 or 48 KB</td>
<td>Efficient physics and ray tracing</td>
</tr>
<tr>
<td>Total Shared Memory</td>
<td>16 KB</td>
<td>16 or 48 KB</td>
<td>More data reuse among threads</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB (TEX read only)</td>
<td>768 KB (all clients read/write)</td>
<td>Greater texture coverage, robust compute performance</td>
</tr>
<tr>
<td>Double Precision Throughput</td>
<td>30 FMAs/clock</td>
<td>256 FMAs/clock</td>
<td>Much higher throughputs for Scientific codes</td>
</tr>
</tbody>
</table>
Instruction Set Architecture

- Enables C++: virtual functions, new/delete, try/catch
- Unified load/store addressing
- 64-bit addressing for large problems
- Optimized for CUDA C, OpenCL & Direct Compute
  - Direct Compute is Microsoft’s general-purpose computing on GPU API
- Enables system call functionality - stdio.h, etc.
Unified Load/Store Addressing

Non-unified Address Space

- Local
  - *p_local
- Shared
  - *p_shared
- Global
  - *p_global

Unified Address Space

- Local
- Shared
- Global

32-bit
40-bit
Instruction Issue and Control Flow

- Decouple internal execution resources
  - Deliver peak IPC on branchy / int-heavy / LD-ST - heavy codes
- Dual issue pipelines select two warps to issue
Caches

• Configurable L1 cache per SM
  – 16KB L1$ / 48KB Shared Memory
  – 48KB L1$ / 16KB Shared Memory

• Shared 768KB L2 cache

• Compute motivation:
  – Caching captures locality, amplifies bandwidth
  – Caching more effective than Shared Memory for irregular or unpredictable access
    • Ray tracing, sparse matrix multiplication, physics kernels …
  – Caching helps latency sensitive cases
GigaThread Hardware Thread Scheduler

• Hierarchically manages tens of thousands of simultaneously active threads
• 10x faster context switching on Fermi
• Concurrent kernel execution
GigaThread Streaming Data Transfer Engine

- Dual DMA engines
- Simultaneous CPU→GPU and GPU→CPU data transfer
- Fully overlapped with CPU/GPU processing
Fermi runs independent kernels in parallel

Concurrent Kernel Execution + Faster Context Switch
Inside Kepler

Manuel Ujaldon
Nvidia CUDA Fellow
Computer Architecture Department
University of Malaga (Spain)

Modified by P. Mordohai
Summary of Features

• Released in 2012
• Architecture: Between 7 and 15 multiprocessors SMX, endowed with 192 cores each.
• Arithmetic: More than 1 TeraFLOP in double precision (64 bits IEEE-754 floating-point format).
  – Specific values depend on the clock frequency for each model (usually, more on GeForces, less on Teslas).
• Major innovations in core design:
  – Dynamic parallelism
  – Thread scheduling (Hyper-Q)
## How the Architecture Scales Up

<table>
<thead>
<tr>
<th>Architecture</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi GF100</th>
<th>Fermi GF104</th>
<th>Kepler GK104</th>
<th>Kepler GK110</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time frame</strong></td>
<td>2006-07</td>
<td>2008-09</td>
<td>2010</td>
<td>2011</td>
<td>2012</td>
<td>2013</td>
</tr>
<tr>
<td><strong>CUDA Compute Capability (CCC)</strong></td>
<td>1.0</td>
<td>1.2</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>N (multiproc.)</td>
<td>16</td>
<td>30</td>
<td>16</td>
<td>7</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>M (cores/multip.)</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>48</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td>Number of cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
<td>336</td>
<td>1536</td>
<td>2880</td>
</tr>
</tbody>
</table>
## Hardware Resources and Peak Performance

<table>
<thead>
<tr>
<th>Tesla card (commercial model)</th>
<th>M2075</th>
<th>M2090</th>
<th>K10</th>
<th>K20</th>
<th>K20X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU generation</strong></td>
<td>Fermi</td>
<td></td>
<td>Kepler</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GPU architecture</strong></td>
<td>GF100</td>
<td></td>
<td>GK104</td>
<td>GK110</td>
<td></td>
</tr>
<tr>
<td><strong>CUDA Compute Capability (CCC)</strong></td>
<td>2.0</td>
<td></td>
<td>3.0</td>
<td>3.5</td>
<td></td>
</tr>
<tr>
<td>GPUs per graphics card</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multiprocessors x (cores/multiproc.)</td>
<td>14 x 32</td>
<td>16 x 32</td>
<td>8 x 192 (x2)</td>
<td>13 x 192</td>
<td>14 x 192</td>
</tr>
<tr>
<td>Total number of cores</td>
<td>448</td>
<td>512</td>
<td>1536 (x2)</td>
<td>2496</td>
<td>2688</td>
</tr>
<tr>
<td>Multiprocessor type</td>
<td>SM</td>
<td></td>
<td>SMX</td>
<td>SMX with dynamic parallelism and HyperQ</td>
<td></td>
</tr>
<tr>
<td>Transistors manufacturing process</td>
<td>40 nm.</td>
<td>40 nm.</td>
<td>28 nm.</td>
<td>28 nm.</td>
<td>28 nm.</td>
</tr>
<tr>
<td>GPU clock frequency (for graphics)</td>
<td>575 MHz</td>
<td>650 MHz</td>
<td>745 MHz</td>
<td>706 MHz</td>
<td>732 MHz</td>
</tr>
<tr>
<td>Core clock frequency (for GPGPU)</td>
<td>1150 MHz</td>
<td>1300 MHz</td>
<td>745 MHz</td>
<td>706 MHz</td>
<td>732 MHz</td>
</tr>
<tr>
<td>Number of single precision cores</td>
<td>448</td>
<td>512</td>
<td>1536 (x2)</td>
<td>2496</td>
<td>2688</td>
</tr>
<tr>
<td>GFLOPS (peak single precision)</td>
<td>1030</td>
<td>1331</td>
<td>2288 (x2)</td>
<td>3520</td>
<td>3950</td>
</tr>
<tr>
<td>Number of double precision cores</td>
<td>224</td>
<td>256</td>
<td>64 (x2)</td>
<td>832</td>
<td>896</td>
</tr>
<tr>
<td>GFLOPS (peak double precision)</td>
<td>515</td>
<td>665</td>
<td>95 (x2)</td>
<td>1170</td>
<td>1310</td>
</tr>
</tbody>
</table>

---

Manuel Ujaldon - Nvidia CUDA Fellow
## Memory Features

<table>
<thead>
<tr>
<th>Tesla card</th>
<th>M2075</th>
<th>M2090</th>
<th>K10</th>
<th>K20</th>
<th>K20X</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit register file / multiprocessor</td>
<td>32768</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>L1 cache + shared memory size</td>
<td>64 KB.</td>
<td>64 KB.</td>
<td>64 KB.</td>
<td>64 KB.</td>
<td>64 KB.</td>
</tr>
<tr>
<td>Width of 32 shared memory banks</td>
<td>32 bits</td>
<td>32 bits</td>
<td>64 bits</td>
<td>64 bits</td>
<td>64 bits</td>
</tr>
<tr>
<td>SRAM clock frequency (same as GPU)</td>
<td>575 MHz</td>
<td>650 MHz</td>
<td>745 MHz</td>
<td>706 MHz</td>
<td>732 MHz</td>
</tr>
<tr>
<td>L1 and shared memory bandwidth</td>
<td>73.6 GB/s</td>
<td>83.2 GB/s</td>
<td>190.7 GB/s</td>
<td>180.7 GB/s</td>
<td>187.3 GB/s</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>768 KB.</td>
<td>768 KB.</td>
<td>768 KB.</td>
<td>1.25 MB.</td>
<td>1.5 MB.</td>
</tr>
<tr>
<td>L2 cache bandwidth (bytes per cycle)</td>
<td>384</td>
<td>384</td>
<td>512</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>L2 on atomic ops. (shared address)</td>
<td>1/9 per clk</td>
<td>1/9 per clk</td>
<td>1 per clk</td>
<td>1 per clk</td>
<td>1 per clk</td>
</tr>
<tr>
<td>L2 on atomic ops. (indep. address)</td>
<td>24 per clk</td>
<td>24 per clk</td>
<td>64 per clk</td>
<td>64 per clk</td>
<td>64 per clk</td>
</tr>
<tr>
<td>DRAM memory width</td>
<td>384 bits</td>
<td>384 bits</td>
<td>256 bits</td>
<td>320 bits</td>
<td>384 bits</td>
</tr>
<tr>
<td>DRAM memory clock (MHz)</td>
<td>2x 1500</td>
<td>2x 1850</td>
<td>2x 2500</td>
<td>2x 2600</td>
<td>2x 2600</td>
</tr>
<tr>
<td>DRAM bandwidth (GB/s, ECC off)</td>
<td>144</td>
<td>177</td>
<td>160 (x2)</td>
<td>208</td>
<td>250</td>
</tr>
<tr>
<td>DRAM generation</td>
<td>GDDR5</td>
<td>GDDR5</td>
<td>GDDR5</td>
<td>GDDR5</td>
<td>GDDR5</td>
</tr>
<tr>
<td>DRAM memory size in Gigabytes</td>
<td>6</td>
<td>6</td>
<td>4 (x2)</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Fermi
Kepler GK110
From SM to SMX in Kepler
Differences in Memory Hierarchy

Kepler Memory Hierarchy

Thread

Shared Memory  L1 Cache

L2 Cache

DRAM

Thread

Shared Memory  L1 Cache  Read-Only Data Cache

L2 Cache

DRAM
New Data Cache

- Additional 48 Kbytes to expand L1 cache size
- Avoids the texture unit
- Allows a global address to be fetched and cached, using a pipeline different from that of L1/shared
- Flexible (does not require aligned accesses)
- Eliminates texture setup
- Managed automatically by compiler ("const__ restrict" indicates eligibility). Next slide shows an example.
How to use Data Cache

- Annotate eligible kernel parameters with "const __restrict"
- Compiler will automatically map loads to use read-only data cache path.

```c
__global__ void saxpy(float x, float y,
   const float * __restrict input,
   float * output)
{
    size_t offset = threadIdx.x +
      (blockIdx.x * blockDim.x);

    // Compiler will automatically use cache for "input"
    output[offset] = (input[offset] * x) + y;
}
```
GPUDirect now supports RDMA [Remote Direct Memory Access]

- This allows direct transfers between GPUs and network devices, for reducing the penalty on the extraordinary bandwidth of GDDR5 video memory.
Relaxing Software Constraints for Massive Parallelism

<table>
<thead>
<tr>
<th>GPU generation</th>
<th>Fermi</th>
<th>Kepler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware model</td>
<td>GF100</td>
<td>GF104</td>
</tr>
<tr>
<td>CUDA CCC</td>
<td>2.0</td>
<td>2.1</td>
</tr>
<tr>
<td>Number of threads / warp (warp size)</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max. number of warps / Multiprocessor</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>Max. number of blocks / Multiprocessor</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Max. number of threads / Block</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Max. number of threads / Multiprocessor</td>
<td>1536</td>
<td>1536</td>
</tr>
</tbody>
</table>

Crucial enhancement for Hyper-Q (see later)
Major Hardware Enhancements

- Large scale computations

<table>
<thead>
<tr>
<th>GPU generation</th>
<th>Fermi</th>
<th>Kepler</th>
<th>Limitation</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware model</td>
<td>GF100</td>
<td>GF104</td>
<td>GK104</td>
<td>GK110</td>
</tr>
<tr>
<td>Compute Capability (CCC)</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>Max. grid size (on X dimension)</td>
<td>$2^{16-1}$</td>
<td>$2^{16-1}$</td>
<td>$2^{32-1}$</td>
<td>$2^{32-1}$</td>
</tr>
</tbody>
</table>

- New architectural features

<table>
<thead>
<tr>
<th>GPU generation</th>
<th>Fermi</th>
<th>Kepler</th>
<th>Limitation</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware model</td>
<td>GF100</td>
<td>GF104</td>
<td>GK104</td>
<td>GK110</td>
</tr>
<tr>
<td>Compute Capability (CCC)</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>Dynamic Parallelism</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Hyper-Q</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Dynamic Parallelism: Hardware (Yes), Problem structure
- Hyper-Q: Hardware (Yes), Thread scheduling
Dynamic Parallelism

• The ability to launch new grids from the GPU:
  – Dynamically: Based on run-time data
  – Simultaneously: From multiple threads at once
  – Independently: Each thread can launch a different grid

Fermi: Only CPU can generate GPU work.

Kepler: GPU can generate work for itself.
Dynamic Parallelism

The pre-Kepler GPU is a co-processor

CPU

GPU

The Kepler GPU is autonomous:
Dynamic parallelism

CPU

GPU

Now programs run faster and are expressed in a more natural way.
Workload Balance

• Plenty of factors, unpredictable at run time, may transform workload balancing among multiprocessors into an impossible goal

• See below the duration of 8 warps on an SM of the G80:
Hyper-Q

- In Fermi, several CPU processes can send thread blocks to the same GPU, but a kernel cannot start its execution until the previous one has finished.
- In Kepler, we can execute simultaneously up to 32 kernels launched from different:
  - MPI processes, CPU threads (POSIX threads) or CUDA streams.
- This increments the % of temporal occupancy on the GPU.
Without Hyper-Q
With Hyper-Q
Six Ways to Improve Code on Kepler

- Dynamic load balancing
- Thread scheduling
- Data-dependent execution
- Recursive parallel algorithms
- Library calls from kernels
- Simplify CPU/GPU divide

Occupancy
Execution
Programmability

Dynamic parallelism and Hyper-Q on Kepler
Dynamic Work Generation

Coarse grid

Higher performance, lower accuracy

Fine grid

Lower performance, higher accuracy

Dynamic grid

Target performance where accuracy is required
Parallelism based on Level of Detail

CUDA until 2012:
- The CPU launches kernels regularly.
- All pixels are treated the same.

CUDA on Kepler:
- The GPU launches a different number of kernels/blocks for each computational region.

Computational power allocated to regions of interest
Grid Management Unit

Fermi

Stream Queue (ordered queues of grids)
- Stream 1: Kernel C, Kernel B, Kernel A
- Stream 2: Kernel R, Kernel Q, Kernel P
- Stream 3: Kernel Z, Kernel Y, Kernel X

Work Distributor
- Tracks blocks issued from grids
- 16 active grids

Kepler GK110

Stream Queue
- Parallel hardware streams

Grid Management Unit
- Pending & Suspended Grids: 1000s of pending grids

Work Distributor
- Actively dispatching grids
- 32 active grids

CUDA Generated Work
- SM, SM, SM, SM
- SMX, SMX, SMX, SMX
Software and Hardware Queues

Fermi:

Up to 16 grids can run at once on GPU hardware.

But CUDA streams multiplex into a single queue.

Chances for overlapping: Only at stream edges.

A -- B -- C
Stream 1

P -- Q -- R
Stream 2

X -- Y -- Z
Stream 3
Software and Hardware Queues

Kepler:

- No inter-stream dependencies
- Up to 32 grids can run at once on GPU hardware
- Concurrency at full-stream level

A--B--C
P--Q--R
X--Y--Z

Streams:
- Stream 1: A -- B -- C
- Stream 2: P -- Q -- R
- Stream 3: X -- Y -- Z
## Instruction Issue and Execution

<table>
<thead>
<tr>
<th></th>
<th><strong>SM-SMX fetch &amp; issue (front-end)</strong></th>
<th><strong>SM-SMX execution (back-end)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fermi (GF100)</strong></td>
<td>Can issue 2 warps, 1 instruction each. Total: <strong>2 warps per cycle.</strong> Active warps: 48 on each SM, chosen from up to 8 blocks. In GTX480: $15 \times 48 = 720$ active warps.</td>
<td>32 cores (1 warp) for &quot;int&quot; and &quot;float&quot;. 16 cores for &quot;double&quot; (1/2 warp). 16 load/store units (1/2 warp). 4 special function units (1/8 warp). A total of up to <strong>4 concurrent warps</strong>.</td>
</tr>
<tr>
<td><strong>Kepler (GK110)</strong></td>
<td>Can issue 4 warps, 2 instructions each. Total: <strong>8 warps per cycle.</strong> Active warps: 64 on each SMX, chosen from up to 16 blocks. In K20: $13 \times 64 = 832$ active warps.</td>
<td>192 cores (6 warps) for &quot;int&quot; and &quot;float&quot;. 64 cores for &quot;double&quot; (2 warps). 32 load/store units (1 warp). 32 special function units (1 warp). A total of up to <strong>10 concurrent warps</strong>.</td>
</tr>
</tbody>
</table>
Data-Dependent Parallelism

• The simplest possible parallel program:
  – Loops are parallelizable
  – Workload is known at compile-time
    \[
    \text{for } i = 1 \text{ to } N \\
    \quad \text{for } j = 1 \text{ to } M \\
    \quad \text{convolution}(i,j);
    \]

• The simplest impossible program:
  – Workload is unknown at compile-time.
  – The challenge is data partitioning
    \[
    \text{for } i = 1 \text{ to } N \\
    \quad \text{for } j = 1 \text{ to } x[i] \\
    \quad \text{convolution}(i,j);
    \]
Data-Dependent Parallelism

• Kepler version:

```c
__global__ void convolution(int x[])
{
    for j = 1 to x[blockIdx]
        // Each block launches x[blockIdx]
        // kernels from GPU
        kernel <<< ... >>> (blockIdx, j)
}
```

// Launch N blocks of 1 thread
// on GPU (rows start in parallel)
convolution <<< N, 1 >>> (x);

• Up to 24 nested loops supported in CUDA 5.0
Recursive Parallel Algorithms prior to Kepler

• Early CUDA programming model did not support recursion at all
• CUDA started to support recursive functions in version 3.1, but they can easily crash if the size of the arguments is large
• A user-defined stack in global memory can be employed instead, but at the cost of a significant performance penalty
• An efficient solution is possible using dynamic parallelism
Parallel Recursion: Quicksort

- Typical divide-and-conquer algorithm hard to do on Fermi
## Quicksort

### Version for Fermi

```c
_global_ void qsort(int *data, int l, int r)
{
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;
    // Partition data around pivot value
    partition(data, l, r, lptr, rptr, pivot);

    // Launch next stage recursively
    int rx = rptr-data; llx = lptr-data;
    if (l < rx)
        qsort<<<...>>>(data, l, rx);
    if (r > llx)
        qsort<<<...>>>(data, llx, r);
}
```

### Version for Kepler

```c
环球_ void qsort(int *data, int l, int r)
{
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;
    // Partition data around pivot value
    partition(data, l, r, lptr, rptr, pivot);

    // Use streams this time for the recursion
    cudaStream_t s1, s2;
    cudaStreamCreateWithFlags(&s1, ...);
    cudaStreamCreateWithFlags(&s2, ...);
    int rx = rptr-data; llx = lptr-data;
    if (l < rx)
        qsort<<<...,0,s1>>>(data, l, rx);
    if (r > llx)
        qsort<<<...,0,s2>>>(data, llx, r);
}
```

Left- and right-hand sorts are serialized

Use separate streams to achieve concurrency.
Quicksort Results
Maxwell
(2nd generation)
Released in 2014

Material by Mark Harris (NVIDIA) and others
Energy Efficiency

Performance per Watt
GTX 680: Kepler  GTX 980: Maxwell
### New Features

<table>
<thead>
<tr>
<th>GPU Parameters</th>
<th>GeForce GTX 680 (Kepler)</th>
<th>GeForce GTX 980 (Maxwell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMs</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>1536</td>
<td>2048</td>
</tr>
<tr>
<td>Base Clock</td>
<td>1006 MHz</td>
<td>1126 MHz</td>
</tr>
<tr>
<td>GPU Boost Clock</td>
<td>1058 MHz</td>
<td>1216 MHz</td>
</tr>
<tr>
<td>GFLOPs</td>
<td>3090</td>
<td>4612</td>
</tr>
<tr>
<td>Texture Units</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Texel fill-rate</td>
<td>128.8 Gigatexels/sec</td>
<td>144.1 Gigatexels/sec</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>6000 MHz</td>
<td>7000 MHz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>192 GB/sec</td>
<td>224 GB/sec</td>
</tr>
<tr>
<td>ROPs</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>512KB</td>
<td>2048KB</td>
</tr>
<tr>
<td>TDP</td>
<td>195 Watts</td>
<td>165 Watts</td>
</tr>
<tr>
<td>Transistors</td>
<td>3.54 billion</td>
<td>5.2 billion</td>
</tr>
<tr>
<td>Die Size</td>
<td>294 mm²</td>
<td>398 mm²</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>28-nm</td>
<td>28-nm</td>
</tr>
</tbody>
</table>
New Features

• Improved instruction scheduling
  – Four warp schedulers per SMM, no shared core functional units

• Increased occupancy
  – Maximum active blocks per SMM has doubled

• Larger dedicated shared memory
  – L1 is now with texture cache

• Faster shared memory atomics

• Broader support for dynamic parallelism
NEXT GENERATION GRAPHICS
Enabling New Algorithms and Superior Image Quality

- Voxel Global Illumination
- Multi Projection
- Conservative Raster
- Shader: Raster Ordered View
- Tiled Resources
- Advanced Sampling
Pascal

Released in 2016
Key New Features

• Smaller manufacturing process
  – 16 nm vs. 28 nm of previous generations
• Much faster memory
• Higher clock frequency
  – 1607 MHz vs. 1216 MHz
• Dynamic load balancing including graphics pipeline
• Page Migration Engine
NVIDIA DGX-1

WORLD’S FIRST DEEP LEARNING SUPERCOMPUTER

Engineered for deep learning | 170TF FP16 | 8x Tesla P100
NVLink hybrid cube mesh | Accelerates major AI frameworks
## "250 SERVERS IN-A-BOX"

<table>
<thead>
<tr>
<th></th>
<th>DUAL XEON</th>
<th>DGX-1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FLOPS (CPU + GPU)</strong></td>
<td>3 TF</td>
<td>170 TF</td>
</tr>
<tr>
<td><strong>AGGREGATE NODE BW</strong></td>
<td>76 GB/s</td>
<td>768 GB/s</td>
</tr>
<tr>
<td><strong>ALEXNET TRAIN TIME</strong></td>
<td>150 HOURS</td>
<td>2 HOURS</td>
</tr>
<tr>
<td><strong>TRAIN IN 2 HOURS</strong></td>
<td>&gt;250 NODES*</td>
<td>1 NODE</td>
</tr>
</tbody>
</table>

*Caffe Training on Multi-node Distributed-memory Systems Based on Intel® Xeon® Processor E5 Family (extrapolated)
Gennady Fedorov (Intel)’s picture Submitted by Gennady Fedorov (Intel), Vadim P. (Intel) on October 29, 2015
AMD RX Vega

- Will be released soon
- 8/16 GB high bandwidth memory (HBM2)
- 14 nm production process
- 12 TFLOPS expected
  - Compared to 11 TFLOPS of NVIDIA GTX Titan X
- 4096 cores
CUDA 4.0
CUDA 4.0: Highlights

**Easier Parallel Application Porting**
- Share GPUs across multiple threads
- Single thread access to all GPUs
- No-copy pinning of system memory
- New CUDA C/C++ features
- Thrust templated primitives library
- NPP image/video processing library
- Layered Textures

**Faster Multi-GPU Programming**
- Unified Virtual Addressing
- NVIDIA GPUDirect™ v2.0
  - Peer-to-Peer Access
  - Peer-to-Peer Transfers
  - GPU-accelerated MPI

**New & Improved Developer Tools**
- Auto Performance Analysis
- C++ Debugging
- GPU Binary Disassembler
- cuda-gdb for MacOS
CUDA 4.0 Release

• March 2011
• Independent software release
• Unlike:
  – CUDA 1.0 released with G80/G9x in 2007 (nearly a year later than the hardware)
  – CUDA 2.0 released for GT200 in 2008
  – CUDA 3.0 released for Fermi in 2009
CUDA 4.0 - Application Porting

• Unified Virtual Addressing

• Faster Multi-GPU Programming
  – NVIDIA GPUDirect 2.0

• Easier Parallel Programming in C++
  – Thrust
Share GPUs across multiple threads

- Easier porting of multi-threaded apps
  - pthreads / OpenMP threads share a GPU
- Launch concurrent kernels from different host threads
  - Eliminates context switching overhead
- New, simple context management APIs
  - Old context migration APIs still supported

Single thread access to all GPUs

- Each host thread can now access all GPUs in the system
  - One thread per GPU limitation removed
- Easier than ever for applications to take advantage of multi-GPU
  - Single-threaded applications can now benefit from multiple GPUs
  - Easily coordinate work across multiple GPUs
No-copy Pinning of System Memory

- Reduce system memory usage and CPU memcpy() overhead
  - Easier to add CUDA acceleration to existing applications
  - Just register malloc’d system memory for async operations and then call cudaMemcpy() as usual

<table>
<thead>
<tr>
<th>Before No-copy Pinning</th>
<th>With No-copy Pinning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra allocation and extra copy required</td>
<td>Just register and go!</td>
</tr>
<tr>
<td>malloc(a)</td>
<td></td>
</tr>
<tr>
<td>cudaMallocHost(b)</td>
<td></td>
</tr>
<tr>
<td>memcpy(b, a)</td>
<td>cudaMemcpy(a)</td>
</tr>
<tr>
<td>cudaMemcpy() to GPU, launch kernels, cudaMemcpy() from GPU</td>
<td></td>
</tr>
<tr>
<td>memcpy(a, b)</td>
<td></td>
</tr>
<tr>
<td>cudaFreeHost(b)</td>
<td>cudaHostUnregister(a)</td>
</tr>
</tbody>
</table>
New CUDA C/C++ Language Features

• C++ new/delete
  – Dynamic memory management

• C++ virtual functions
  – Easier porting of existing applications

• Inline PTX
  – Enables assembly-level optimization
GPU-Accelerated Image Processing

- NVIDIA Performance Primitives (NPP) library
  - 10x to 36x faster image processing
  - Initial focus on imaging and video related primitives
    - Data exchange and initialization
    - Color conversion
    - Threshold and compare operations
    - Statistics
    - Filter functions
    - Geometry transforms
    - Arithmetic and logical operations
    - JPEG
Layered Textures - Faster Image Processing

- Ideal for processing multiple textures with same size/format
  - Large sizes supported on Tesla T20 (Fermi) GPUs (up to 16k x 16k x 2k)
  - e.g. Medical Imaging, Terrain Rendering (flight simulators), etc.

- Faster Performance
  - Reduced CPU overhead: single binding for entire texture array
  - Faster than 3D Textures: more efficient filter caching
  - Fast interop with OpenGL / Direct3D for each layer
  - No need to create/manage a texture atlas

- No sampling artifacts
  - Linear/Bilinear filtering applied only within a layer
NVIDIA GPUDirect: Towards Eliminating the CPU Bottleneck

**Version 1.0**
*for applications that communicate over a network*

- Direct access to GPU memory for 3rd party devices
- Eliminates unnecessary sys mem copies & CPU overhead
- Supported by Mellanox and Qlogic
- Up to 30% improvement in communication performance

**Version 2.0**
*for applications that communicate within a node*

- Peer-to-Peer memory access, transfers & synchronization
- Less code, higher programmer productivity
Before GPUDirect 2.0

Two copies required
GPUDirect 2.0: Peer-to-Peer Communication
Only one copy required
GPUDirect 2.0: Peer-to-Peer Communication

• Direct communication between GPUs
  – Faster - no system memory copy overhead
  – More convenient multi-GPU programming

• Direct Transfers
  – Copy from GPU0 memory to GPU1 memory
  – Works transparently with UVA

• Direct Access
  – GPU0 reads or writes GPU1 memory (load/store)

• Supported on Tesla 20-series and other Fermi GPUs
  – 64-bit applications on Linux and Windows
Unified Virtual Addressing

- No UVA: Multiple Memory Spaces
- UVA: Single Address Space
Unified Virtual Addressing

- One address space for all CPU and GPU memory
  - Determine physical memory location from pointer value
  - Enables libraries to simplify their interfaces (e.g. cudaMemcpy)
- Supported on Tesla 20-series and other Fermi GPUs

<table>
<thead>
<tr>
<th>Before UVA</th>
<th>With UVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate options for each permutation</td>
<td>One function handles all cases</td>
</tr>
<tr>
<td>cudaMemcpyHostToDevice</td>
<td>cudaMemcpyDefault (data location becomes an implementation detail)</td>
</tr>
<tr>
<td>cudaMemcpyHostToDevice</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyDeviceToDevice</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyDeviceToDeviceToHost</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyDeviceToDeviceToDevice</td>
<td></td>
</tr>
</tbody>
</table>
New Developer Tools

• Auto Performance Analysis: Visual Profiler
  – Identify limiting factor
  – Analyze instruction throughput
  – Analyze memory throughput
  – Analyze kernel occupancy

• C++ Debugging
  – cuda-gdb for MacOS

• GPU Binary Disassembler
CUDA 5.0

Mark Harris
Chief Technologist, GPU Computing
Open Source LLVM Compiler

- Provides ability for anyone to add CUDA to new languages and processors
NVIDIA Nsight, Eclipse Edition

CUDA-Aware Editor
- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Integrated code samples & docs

Nsight Debugger
- Simultaneously debug of CPU and GPU
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging

Nsight Profiler
- Quickly identifies performance issues
- Integrated expert system
- Automated analysis
- Source line correlation

For Linux and Mac OS
CUDA 4: Whole-Program Compilation & Linking
CUDA 5: GPU Library Object Linking

- Separate compilation allows building independent object files
- CUDA 5 can link multiple object files into one program
- Can also combine object files into static libraries
  - Link and externally call *device* code

```
main.cpp + a.cu b.cu c.cu → program.exe
```

```
a.o b.o c.o
```
CUDA 5: GPU Library Object Linking

- Enables 3rd party closed-source device libraries
- User-defined device callback functions
CUDA 5.0: Run-time Syntax and Semantics

```c
__device__ float buf[1024];
__global__ void dynamic(float *data)
{
    int tid = threadIdx.x;
    if (tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];
__syncthreads();

    if (tid == 0) {
        launchKernel<<<128,256>>>(buf);
        cudaDeviceSynchronize();
    }
__syncthreads();

    if (tid == 0) {
        cudaMemcpyAsync(data, buf, 1024);
        cudaDeviceSynchronize();
    }
}
```

This launch is per-thread
CUDA 5.0: Sync. all launches within my block
idle threads wait for the others here
CUDA 5.0: Only async. launches are allowed on data gathering
CUDA 6.0

Manuel Ujaldon
Nvidia CUDA Fellow
Computer Architecture
Department
University of Malaga (Spain)
CUDA 6 Highlights

• Unified Memory:
  – CPU and GPU can share data without much programming effort
• Extended Library Interface (XT) and Drop-in Libraries:
  – Libraries much easier to use
• GPUDirect RDMA:
  – A key achievement in multi-GPU environments
• Developer tools:
  – Visual Profiler enhanced with:
    • Side-by-side source and disassembly view showing.
    • New analysis passes (per SM activity level), generates a kernel analysis report.
• Multi-Process Server (MPS) support in nvprof and cuda-memcheck
• Nsight Eclipse Edition supports remote development (x86 and ARM)
CUDA 6.0: Performance Improvements in Key Use Cases

• Kernel launch
• Repeated launch of the same set of kernels
• cudaDeviceSynchronize()
• Back-to-back grids in a stream
Unified Memory

- CPU
  - DDR3 (Dual-, tri- or quad-channel (~100 GB/s.))
  - GDDR5 (PCI-express (~10 GB/s.))
  - Main memory
- GPU
  - 256, 320, 384 bits (~300 GB/s.)
  - Video memory
- CPU
  - Kepler+ GPU
  - DDR3
  - Unified memory
  - GDDR5
Unified Memory Contributions

• Creates pool of managed memory between CPU and GPU

• Simpler programming and memory model:
  – Single pointer to data, accessible anywhere
  – Eliminate need for cudaMemcpy(), use cudaMemcpyManaged()
  – No need for deep copies

• Performance through data locality:
  – Migrate data to accessing processor
  – Guarantee global coherency
  – Still allows cudaMemcpyAsync() hand tuning
## Memory Types

<table>
<thead>
<tr>
<th></th>
<th>Zero-Copy (pinned memory)</th>
<th>Unified Virtual Addressing</th>
<th>Unified Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA call</td>
<td>cudaMallocHost(&amp;A, 4);</td>
<td>cudaMalloc(&amp;A, 4);</td>
<td>cudaMallocManaged(&amp;A, 4);</td>
</tr>
<tr>
<td>Allocation fixed in</td>
<td>Main memory (DDR3)</td>
<td>Video memory (GDDR5)</td>
<td>Both</td>
</tr>
<tr>
<td>Local access for</td>
<td>CPU</td>
<td>Home GPU</td>
<td>CPU and home GPU</td>
</tr>
<tr>
<td>PIC-e access for</td>
<td>All GPUs</td>
<td>Other GPUs</td>
<td>Other GPUs</td>
</tr>
<tr>
<td>Other features</td>
<td>Avoid swapping to disk</td>
<td>No CPU access</td>
<td>On access CPU/GPU migration</td>
</tr>
<tr>
<td>Coherency</td>
<td>At all times</td>
<td>Between GPUs</td>
<td>Only at launch &amp; sync.</td>
</tr>
<tr>
<td>Full support in</td>
<td>CUDA 2.2</td>
<td>CUDA 1.0</td>
<td>CUDA 6.0</td>
</tr>
</tbody>
</table>
Additions to the CUDA API

• New call: `cudaMallocManaged()`
  – Drop-in replacement for `cudaMalloc()` allocates managed memory
  – Returns pointer accessible from both Host and Device

• New call: `cudaStreamAttachMemAsync()`
  – Manages concurrency in multi-threaded CPU applications

• New keyword: `__managed__`
  – Declares global-scope migratable device variable
  – Symbol accessible from both GPU and CPU code
Code without Unified Memory

```c
void launch(dataElem *elem) {
    dataElem *g_elem;
    char *g_text;

    int txtlen = strlen(elem->text);

    // Allocate storage for struct and text
    cudaMalloc(&g_elem, sizeof(dataElem));
    cudaMalloc(&g_text, txtlen);

    // Copy up each piece separately, including new "text" pointer value
    cudaMemcpy(g_elem, elem, sizeof(dataElem));
    cudaMemcpy(g_text, elem->text, txtlen);
    cudaMemcpy((g_elem->text), &g_text, sizeof(g_text));

    // Finally we can launch our kernel, but
    // CPU and GPU use different copies of "elem"
    kernel<<< ... >>>(g_elem);
}
```
Code with Unified Memory

• What remains the same:
  – Data movement
  – GPU accesses a local copy of text

• What has changed:
  – Programmer sees a single pointer
  – CPU and GPU both reference the same object
  – There is coherence

```c
void launch(dataElem *elem) {
    kernel<<< ... >>>(elem);
}
```
CUDA 7.0

By Mark Harris
NVIDIA
New Features: C++11

• C++11 features on device including:
  – auto,
  – lambda,
  – variadic templates,
  – rvalue references,
  – range-based for loops
Example

#include <initializer_list>
#include <iostream>
#include <cstring>

// Generic parallel find routine. Threads search through the array in parallel. A thread returns the index of the first value it finds that satisfies predicate `p`, or -1.

template <typename T, typename Predicate>
__device__ int find(T *data, int n, Predicate p)
{
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
         i < n;
         i += blockDim.x * gridDim.x)
    {
        if (p(data[i])) return i;
    }
    return -1;
}
// Use find with a lambda function that searches for x, y, z
// or w. Note the use of range-based for loop and
// initializer_list inside the functor, and auto means we
// don't have to know the type of the lambda or the array
__global__
void xyzw_frequency(unsigned int *count, char *data, int n)
{
    auto match_xyzw = [] (char c) {
        const char letters[] = { 'x','y','z','w' };  
        for (const auto x : letters) 
            if (c == x) return true;
        return false;
    };

    int i = find(data, n, match_xyzw);

    if (i >= 0) atomicAdd(count, 1);
}
int main(void)
{
    char text[] = "zebra xylophone wax";
    char *d_text;

    cudaMemcpy(&d_text, text, sizeof(text), cudaMemcpyHostToDevice);

    unsigned int *d_count;
    cudaMemcpy(&d_count, text, sizeof(unsigned int), cudaMemcpyHostToDevice);

    xyzw_frequency<<<1, 64>>>(d_count, d_text, strlen(text));

    cudaMemcpy(&count, d_count, sizeof(unsigned int), cudaMemcpyDeviceToHost);

    std::cout << count << " instances of 'x', 'y', 'z', 'w'" << " in " << text << std::endl;

    cudaFree(d_text);
    cudaFree(d_count);

    return 0;
}
Other Features

• Thrust version 1.8
  – Thrust algorithms can now be invoked from the device

• cuSOLVER, cuFFT
  – cuSolver library is a high-level package based on the cuBLAS and cuSPARSE libraries

• Runtime compilation
  – No need to generate multiple optimized kernels at compile time
CUDA 8.0

By Milind Kukanur
NVIDIA
What’s New

**PASCAL SUPPORT**
- New Architecture
- NVLINK
- HBM2 Stacked Memory
- Page Migration Engine

**UNIFIED MEMORY**
- Larger Datasets
- Demand Paging
- New Tuning APIs
- Data Coherence & Atomics

**LIBRARIES**
- New nvGRAPH library
- cuBLAS improvements for Deep Learning

**DEVELOPER TOOLS**
- Critical Path Analysis
- 2x Faster Compile Time
- OpenACC Profiling
- Debug CUDA Apps on Display GPU
Unified Memory

• Oversubscribe GPU memory, up to system memory size

```c
void foo() {
    // Allocate 64 GB
    char *data;
    size_t size = 64*1024*1024*1024;
    cudaMallocManaged(&data, size);
}
```
Unified Memory

```c
__global__ void mykernel(char *data) {
    data[1] = 'g';
}

void foo() {
    char *data;
    cudaMallocManaged(&data, 2);

    mykernel<<<...>>>(data);
    // no synchronize here
    data[0] = 'c';

    cudaFree(data);
}
```