CS 677: Parallel Programming for Many-core Processors
Lecture 12

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

• Parallel Sorting

• Hardware Developments
• Developments in CUDA

• Introduction to OpenMP
OpenCL Sorting

Eric Bainville - June 2011
Parallel Selection Sort

__kernel void ParallelSelection(__global const data_t * in,__global data_t * out)
{
    int i = get_global_id(0); // current thread
    int n = get_global_size(0); // input size
    data_t iData = in[i];
    uint iKey = keyValue(iData);
    // Compute position of in[i] in output
    int pos = 0;
    for (int j=0;j<n;j++)
    {
        uint jKey = keyValue(in[j]); // broadcasted
        // in[j] < in[i] ?
        bool smaller = (jKey < iKey) || (jKey == iKey && j < i);
        pos += (smaller)?1:0;
    }
    out[pos] = iData;
}
Parallel Selection Sort

• Very ineffective
• \(2N+N^2\) accesses to global memory. Why?

• A.k.a. Parallel Rank Sort
  – Effective on multi-processor system with high-bandwidth memory
Parallel Selection Sort, blocks

__kernel void ParallelSelection_Blocks(__global const data_t * in,__global data_t * out,__local uint * aux)
{
    int i = get_global_id(0); // current thread
    int n = get_global_size(0); // input size
    int wg = get_local_size(0); // workgroup size

    data_t iData = in[i]; // input record for current thread
    uint iKey = keyValue(iData); // input key for current thread
    int blockSize = BLOCK_FACTOR * wg; // block size
// Compute position of iKey in output
int pos = 0;
// Loop on blocks of size BLOCKSIZE keys (BLOCKSIZE must divide N)
for (int j=0;j<n;j+=blockSize)
{
    // Load BLOCKSIZE keys using all threads (BLOCK_FACTOR values per thread)
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int index=get_local_id(0);index<blockSize;index+=wg)
        aux[index] = keyValue(in[j+index]);
    barrier(CLK_LOCAL_MEM_FENCE);

    // Loop on all values in AUX
    for (int index=0;index<blockSize;index++)
    {
        uint jKey = aux[index]; // broadcasted, local memory
        // in[j] < in[i] ?
        bool smaller = (jKey < iKey) || (jKey == iKey && (j+index) < i);
        pos += (smaller)?1:0;
    }

    out[pos] = iData;
}

Compare-and-Exchange Sorting

Fikret Ercal (Missouri University of Science and Technology)
and
Fernando Silva (University of Porto)
Odd-Even Transposition Sort

Parallel time complexity: \( T_{\text{par}} = O(N) \) (for \( P=N \))
Odd-Even Transposition Sort (N>>P)

Each PE gets $\frac{N}{P}$ numbers. First, PEs sort $\frac{N}{P}$ locally, then they run odd-even trans. algorithm each time doing a merge-split for $2\frac{N}{P}$ numbers.

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 7 12</td>
<td>8 5 4</td>
<td>6 1 3</td>
<td>9 2 10</td>
</tr>
</tbody>
</table>

Local sort

| 7 12 13 | 4 5 8   | 1 3 6   | 2 9 10 |

O-E

| 4 5 7   | 8 12 13 | 1 2 3   | 6 9 10 |

E-O

| 4 5 7   | 1 2 3   | 8 12 13 | 6 9 10 |

O-E

| 1 2 3   | 4 5 7   | 6 8 9   | 10 12 13 |

E-O

SORTED: 1 2 3 4 5 6 7 8 9 10 12 13
Merge-Split

Sorted lists

Even indices

Odd indices

Merge

Compare and exchange

Final sorted list

1 2 3 4 5 6 7 8
Bitonic Mergesort

A bitonic sequence is defined as a list with no more than one LOCAL MAXIMUM and no more than one LOCAL MINIMUM. (Endpoints must be considered - wraparound)

(a) Single maximum
(b) Single maximum and single minimum
Binary Split

1. Divide the bitonic list into two equal halves.
2. Compare-Exchange each item on the first half with the corresponding item in the second half.

Result:
Two bitonic sequences where the numbers in one sequence are all less than the numbers in the other sequence.
### Repeated Application of Binary Split

Bitonic list:

```
| 24 20 15 9 4 2 5 8 | 10 11 12 13 22 30 32 45 |
```

Result after Binary-split:

```
| 10 11 12 9 4 2 5 8 | 24 20 15 13 22 30 32 45 |
```

If you keep applying the BINARY-SPLIT to each half repeatedly, you will get a SORTED LIST!

```
| 10 11 12 9 4 2 5 8 | 24 20 15 13 22 30 32 45 |
| 4 2 5 8 10 11 12 9 | 22 20 15 13 24 30 32 45 |
| 4 2 5 8 10 9 12 11 | 15 13 22 20 24 30 32 45 |
| 2 4 5 8 9 10 11 12 | 13 15 20 22 24 30 32 45 |
```

Q: How many parallel steps does it take to sort?
A: \( \log n \)
Sorting a Bitonic Sequence

- Compare-and-exchange moves smaller numbers of each pair to left and larger numbers of pair to right.
- Given a bitonic sequence, recursively performing ‘binary split’ will sort the list.
Sorting an Arbitrary Sequence

• To sort an unordered sequence, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers.

• A sequence of length 2 is a bitonic sequence.

• A bitonic sequence of length 4 can be built by sorting the first two elements using a positive bitonic merge and the next two using a negative bitonic merge.
Sorting an Arbitrary Sequence

• By a compare-and-exchange operation, pairs of adjacent numbers form increasing sequences and decreasing sequences. Pairs form a bitonic sequence of twice the size of each original sequences.

• By repeating this process, bitonic sequences of larger and larger lengths obtained.

• In the final step, a single bitonic sequence is sorted into a single increasing sequence.
Bitonic Mergesort

- Whenever two numbers reach the two ends of an arrow, they are compared to ensure that the arrow points toward the larger number.
- If they are out of order, they are swapped.

Form bitonic lists of four numbers

Form bitonic list of eight numbers

Sort bitonic list

Compare and exchange

Lower Higher
```python
def bitonic_sort(up, x):
    if len(x) <= 1:
        return x
    else:
        first = bitonic_sort(True, x[:len(x) // 2])
        second = bitonic_sort(False, x[len(x) // 2:]
        return bitonic_merge(up, first + second)

def bitonic_merge(up, x):
    # assume input x is bitonic, and sorted list is returned
    if len(x) == 1:
        return x
    else:
        bitonic_compare(up, x)
        first = bitonic_merge(up, x[:len(x) // 2])
        second = bitonic_merge(up, x[len(x) // 2:]
        return first + second

def bitonic_compare(up, x):
    dist = len(x) / 2
    for i in range(dist):
        if (x[i] > x[i + dist]) == up:
            x[i], x[i + dist] = x[i + dist], x[i]  # swap
```

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

<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>256KB (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>

- **Reliability: ECC**
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
  - 32-bit

Unified Address Space

- Local
- Shared
- Global
  - 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

<table>
<thead>
<tr>
<th>Warp Scheduler</th>
<th>Instruction Dispatch Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 8 instruction 11</td>
<td>Warp 9 instruction 11</td>
</tr>
<tr>
<td>Warp 2 instruction 42</td>
<td>Warp 3 instruction 33</td>
</tr>
<tr>
<td>Warp 14 instruction 95</td>
<td>Warp 15 instruction 95</td>
</tr>
</tbody>
</table>
| ... | ...
| Warp 8 instruction 12 | Warp 9 instruction 12 |
| Warp 14 instruction 96 | Warp 3 instruction 34 |
| Warp 2 instruction 43 | Warp 15 instruction 96 |
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

Serial Kernel Execution

Parallel Kernel Execution

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>CUDA Compute Capability (CCC)</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 (multiprocs.)</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>
Fermi
Kepler GK110
From SM to SMX in Kepler
Differences in Memory Hierarchy

Thread

Shared Memory

L1 Cache

L2 Cache

DRAM

Kepler Memory Hierarchy

Thread

Shared Memory

L1 Cache

L2 Cache

Read-Only Data 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 Compute Capability (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>$2^{16}$</td>
<td>$2^{16}$</td>
<td>$2^{32}$</td>
<td>$2^{32}$</td>
</tr>
<tr>
<td>Software</td>
<td>Problem size</td>
<td></td>
<td></td>
<td></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>
<tr>
<td>Hardware</td>
<td></td>
<td></td>
<td>Problem structure</td>
<td></td>
</tr>
<tr>
<td>Thread scheduling</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

The Kepler GPU is autonomous:
Dynamic parallelism

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.

![Diagram showing comparison between Fermi and Kepler](image-url)
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

Dynamic parallelism and Hyper-Q on Kepler

Occupancy

Execution

Programmability
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**
  - C, B, A
  - R, Q, P
  - Z, Y, X

- **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

- **Parallel hardware streams**
  - Allows suspending of grids
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.

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

<table>
<thead>
<tr>
<th></th>
<th>SM-SMX fetch &amp; issue (front-end)</th>
<th>SM-SMX execution (back-end)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fermi (GF100)</strong></td>
<td>Can issue 2 warps, 1 instruction each.</td>
<td>32 cores (1 warp) for &quot;int&quot; and &quot;float&quot;.</td>
</tr>
<tr>
<td></td>
<td>Total: <strong>2 warps per cycle.</strong></td>
<td>16 cores for &quot;double&quot; (1/2 warp).</td>
</tr>
<tr>
<td></td>
<td>Active warps: 48 on each SM,</td>
<td>16 load/store units (1/2 warp).</td>
</tr>
<tr>
<td></td>
<td>chosen from up to 8 blocks.</td>
<td>4 special function units (1/8 warp).</td>
</tr>
<tr>
<td></td>
<td>In GTX480: 15 * 48 = 720 active warps.</td>
<td>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.</td>
<td>192 cores (6 warps) for &quot;int&quot; and &quot;float&quot;.</td>
</tr>
<tr>
<td></td>
<td>Total: <strong>8 warps per cycle.</strong></td>
<td>64 cores for &quot;double&quot; (2 warps).</td>
</tr>
<tr>
<td></td>
<td>Active warps: 64 on each SMX,</td>
<td>32 load/store units (1 warp).</td>
</tr>
<tr>
<td></td>
<td>chosen from up to 16 blocks.</td>
<td>32 special function units (1 warp).</td>
</tr>
<tr>
<td></td>
<td>In K20: 13 * 64 = 832 active warps.</td>
<td>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
  
  for i = 1 to N
    for j = 1 to M
      convolution(i,j);

• The simplest impossible program:
  – Workload is unknown at compile-time.
  – The challenge is data partitioning
  
  for i = 1 to N
    for j = 1 to x[i]
      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 *lpotr = data+1, *rpotr = data+r;
    // Partition data around pivot value
    partition(data, l, r, lpotr, rpotr, pivot);

    // Launch next stage recursively
    int rx = rpotr-data; lx = lpotr-data;
    if (l < rx)
        qsort<<<...>>>(data, l, rx);
    if (r > lx)
        qsort<<<...>>>(data, lx, r);
}
```

left- and right-hand sorts are serialized

## Version for Kepler

```c
__global__ void qsort(int *data, int l, int r)
{
    int pivot = data[0];
    int *lpotr = data+1, *rpotr = data+r;
    // Partition data around pivot value
    partition(data, l, r, lpotr, rpotr, pivot);

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

Use separate streams to achieve concurrency
Quicksort Results

![Quicksort chart](chart.png)

Relative Sorting Performance

Problem Size (Million of Elements)

- Without Dynamic Parallelism
- With Dynamic Parallelism

browns
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</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>4612³</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 (Maxwell SM), 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
Volta

Released in 2017
Key New Features

• Up to 640 Tensor Cores for deep learning
  – Multiply and add floating point matrices (64 operations per clock)
  – Over 125 TFLOPS (5x more than Pascal)

• Next generation NVLink doubles bandwidth (up to 300 GB/s)

• 84 SMs

• Simultaneous execution of FP32 and INT32 operations
<table>
<thead>
<tr>
<th>Tesla Product</th>
<th>Tesla K40</th>
<th>Tesla M40</th>
<th>Tesla P100</th>
<th>Tesla V100</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>GK180 (Kepler)</td>
<td>GM200 (Maxwell)</td>
<td>GP100 (Pascal)</td>
<td>GV100 (Volta)</td>
</tr>
<tr>
<td>SMs</td>
<td>15</td>
<td>24</td>
<td>56</td>
<td>80</td>
</tr>
<tr>
<td>TPCs</td>
<td>15</td>
<td>24</td>
<td>28</td>
<td>40</td>
</tr>
<tr>
<td>FP32 Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>FP32 Cores / GPU</td>
<td>2880</td>
<td>3072</td>
<td>3584</td>
<td>5120</td>
</tr>
<tr>
<td>FP64 Cores / SM</td>
<td>64</td>
<td>4</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>FP64 Cores / GPU</td>
<td>960</td>
<td>96</td>
<td>1792</td>
<td>2560</td>
</tr>
<tr>
<td>Tensor Cores / SM</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>8</td>
</tr>
<tr>
<td>Tensor Cores / GPU</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>640</td>
</tr>
<tr>
<td>GPU Boost Clock</td>
<td>810/875 MHz</td>
<td>1114 MHz</td>
<td>1480 MHz</td>
<td>1530 MHz</td>
</tr>
<tr>
<td>Peak FP32 TFLOPS¹</td>
<td>5</td>
<td>6.8</td>
<td>10.6</td>
<td>15.7</td>
</tr>
<tr>
<td>Peak FP64 TFLOPS¹</td>
<td>1.7</td>
<td>0.21</td>
<td>5.3</td>
<td>7.8</td>
</tr>
<tr>
<td>Peak Tensor TFLOPS¹</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>125</td>
</tr>
<tr>
<td>Texture Units</td>
<td>240</td>
<td>192</td>
<td>224</td>
<td>320</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>384-bit GDDR5</td>
<td>384-bit GDDR5</td>
<td>4096-bit HBM2</td>
<td>4096-bit HBM2</td>
</tr>
<tr>
<td>Memory Size</td>
<td>Up to 12 GB</td>
<td>Up to 24 GB</td>
<td>16 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1536 KB</td>
<td>3072 KB</td>
<td>4096 KB</td>
<td>6144 KB</td>
</tr>
<tr>
<td>Shared Memory Size / SM</td>
<td>16 KB/32 KB/48 KB</td>
<td>96 KB</td>
<td>64 KB</td>
<td>Configurable up to 96 KB</td>
</tr>
<tr>
<td>Register File Size / SM</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256KB</td>
</tr>
<tr>
<td>Register File Size / GPU</td>
<td>3840 KB</td>
<td>6144 KB</td>
<td>14336 KB</td>
<td>20480 KB</td>
</tr>
<tr>
<td>TDP</td>
<td>235 Watts</td>
<td>250 Watts</td>
<td>300 Watts</td>
<td>300 Watts</td>
</tr>
<tr>
<td>Transistors</td>
<td>7.1 billion</td>
<td>8 billion</td>
<td>15.3 billion</td>
<td>21.1 billion</td>
</tr>
<tr>
<td>GPU Die Size</td>
<td>551 mm²</td>
<td>601 mm²</td>
<td>610 mm²</td>
<td>815 mm²</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>28 nm</td>
<td>28 nm</td>
<td>16 nm FinFET+</td>
<td>12 nm FFN</td>
</tr>
</tbody>
</table>

¹ Peak TFLOPS rates are based on GPU Boost Clock
Turing

Released in 2018
Key New Features

• CUDA, Ray-tracing and Tensor cores
  – 14.2 TFLOPS of FP32 performance, 113.8 Tensor TFLOPS and 10 Giga Rays/sec
• Up to 24 GB of RAM in Titan RTX

• Independent integer and floating-point datapaths and unified shared memory, texture caching and memory load caching lead to 50% performance improvement per core
Turing Tensor Cores
Memory Compression

- Several lossless memory compression techniques to reduce bandwidth demands
- Improvements over Pascal
Reflections Demo
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>FLOPS (CPU + GPU)</td>
<td>3 TF</td>
<td>170 TF</td>
</tr>
<tr>
<td>AGGREGATE NODE BW</td>
<td>76 GB/s</td>
<td>768 GB/s</td>
</tr>
<tr>
<td>ALEXNET TRAIN TIME</td>
<td>150 HOURS</td>
<td>2 HOURS</td>
</tr>
<tr>
<td>TRAIN IN 2 HOURS</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
# System Specifications

<table>
<thead>
<tr>
<th><strong>GPUs</strong></th>
<th>8X Tesla V100</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>1 petaFLOPS</td>
</tr>
<tr>
<td><strong>(Mixed Precision)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>GPU Memory</strong></td>
<td>256 GB total system</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>Dual 20-Core Intel Xeon E5-2698 v4 2.2 GHz</td>
</tr>
<tr>
<td><strong>NVIDIA CUDA® Cores</strong></td>
<td>40,960</td>
</tr>
<tr>
<td><strong>NVIDIA Tensor Cores</strong></td>
<td>5,120</td>
</tr>
<tr>
<td><strong>(on V100 based systems)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Power Requirements</strong></td>
<td>3,500 W</td>
</tr>
<tr>
<td><strong>System Memory</strong></td>
<td>512 GB 2,133 MHz DDR4 RDIMM</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>4X 1.92 TB SSD RAID 0</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>Dual 10 GbE, 4 IB EDR</td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
<td>Canonical Ubuntu, Red Hat Enterprise Linux</td>
</tr>
<tr>
<td><strong>System Weight</strong></td>
<td>134 lbs</td>
</tr>
<tr>
<td><strong>System Dimensions</strong></td>
<td>866 D x 444 W x 131 H (mm)</td>
</tr>
<tr>
<td><strong>Packing Dimensions</strong></td>
<td>1,180 D x 730 W x 284 H (mm)</td>
</tr>
<tr>
<td><strong>Operating Temperature Range</strong></td>
<td>5–35 °C</td>
</tr>
</tbody>
</table>
NVIDIA DGX-2

The world’s most powerful AI system for the most complex AI challenges.
# SYSTEM SPECIFICATIONS

<table>
<thead>
<tr>
<th><strong>GPUs</strong></th>
<th>16X NVIDIA® Tesla V100</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU Memory</strong></td>
<td>512GB total</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>2 petaFLOPS</td>
</tr>
<tr>
<td><strong>NVIDIA CUDA® Cores</strong></td>
<td>81920</td>
</tr>
<tr>
<td><strong>NVIDIA Tensor Cores</strong></td>
<td>10240</td>
</tr>
<tr>
<td><strong>NVSwitches</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>Maximum Power Usage</strong></td>
<td>10kW</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>Dual Intel Xeon Platinum 8168, 2.7 GHz, 24-cores</td>
</tr>
<tr>
<td><strong>System Memory</strong></td>
<td>1.5TB</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td>8X 100Gb/sec Infiniband/100GigE Dual 10/25/40/50/100GbE</td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>OS: 2X 960GB NVME SSDs Internal Storage: 30TB (8X 3.84TB) NVME SSDs</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>Ubuntu Linux OS Red Hat Enterprise Linux OS See Software stack for details</td>
</tr>
<tr>
<td><strong>System Weight</strong></td>
<td>360 lbs (163.29 kgs)</td>
</tr>
<tr>
<td><strong>Packaged System Weight</strong></td>
<td>400lbs (181.44kgs)</td>
</tr>
<tr>
<td><strong>System Dimensions</strong></td>
<td>Height: 17.3 in (440.0 mm) Width: 19.0 in (482.3 mm) Length: 31.3 in (795.4 mm) - No Front Bezel 32.8 in (834.0 mm) - With Front Bezel</td>
</tr>
<tr>
<td><strong>Operating Temperature Range</strong></td>
<td>5°C to 35°C (41°F to 95°F)</td>
</tr>
</tbody>
</table>
AMD RX Vega

- 8 GB high bandwidth memory (HBM2)
  - 14 nm production process
- 4096 cores
- 12.7 TFLOPS
  - Compared to 11 TFLOPS of NVIDIA GTX Titan X and 15.7 TFLOPS of NVIDIA GV100 (Volta)
AMD RADEON VII

- 16 GB high bandwidth memory (HBM2)
  - 7 nm production process
- 3840 cores
- 13.2 billion transistors
- 13.8 TFLOPS
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
Easier Porting of Existing Applications

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
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 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
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</td>
</tr>
<tr>
<td>cudaMemcpyHostToDeviceToHost</td>
<td>(data location becomes an implementation detail)</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

main.cpp + a.cu + b.cu + c.cu -> program.exe

Include files together to build
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
CUDA 5: GPU Library Object Linking

- Enables 3rd party closed-source device libraries
- User-defined device callback functions
__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();
    }
}
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

Dual-, tri- or quad-channel (~100 GB/s.)

DDR3 (~10 GB/s.)

Main memory

GPU

256, 320, 384 bits (~300 GB/s.)

GDDR5

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 cudaMallocManaged()
  – 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><strong>CUDA call</strong></td>
<td>cudaMallocHost(&amp;A, 4);</td>
<td>cudaMalloc(&amp;A, 4);</td>
<td>cudaMallocManaged(&amp;A, 4);</td>
</tr>
<tr>
<td><strong>Allocation fixed in</strong></td>
<td>Main memory (DDR3)</td>
<td>Video memory (GDDR5)</td>
<td>Both</td>
</tr>
<tr>
<td><strong>Local access for</strong></td>
<td>CPU</td>
<td>Home GPU</td>
<td>CPU and home GPU</td>
</tr>
<tr>
<td><strong>PIC-e access for</strong></td>
<td>All GPUs</td>
<td>Other GPUs</td>
<td>Other GPUs</td>
</tr>
<tr>
<td><strong>Other features</strong></td>
<td>Avoid swapping to disk</td>
<td>No CPU access</td>
<td>On access CPU/GPU migration</td>
</tr>
<tr>
<td><strong>Coherency</strong></td>
<td>At all times</td>
<td>Between GPUs</td>
<td>Only at launch &amp; sync.</td>
</tr>
<tr>
<td><strong>Full support in</strong></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 strlen = strlen(elem->text);

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

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

```cpp
#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;

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

    unsigned int *d_count;
    cudaMalloc(&d_count, sizeof(unsigned int));
    cudaMemcpy(d_count, 0, sizeof(unsigned int), cudaMemcpyHostToDevice);

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

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

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

    cudaFree(d_count);
    cudaFree(d_text);

    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

__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);
}
CUDA 9.0

By Mark Harris
NVIDIA
New Features

• Support for Volta
• Cooperative groups
• Tensor Core API
• New Visual Profiler
• Support for C++ 14
Cooperative Groups

- Ability to define groups of threads explicitly at sub-block and multiblock granularities

```cpp
__global__ void cooperative_kernel(...)
{

    // obtain default "current thread block" group
    thread_group my_block = this_thread_block();

    // subdivide into 32-thread, tiled subgroups
    // Tiled subgroups evenly partition a parent group into
    // adjacent sets of threads - in this case each one warp in size
    thread_group my_tile = tiled_partition(my_block, 32);

    // This operation will be performed by only the
    // first 32-thread tile of each block
    if (my_block.thread_rank() < 32) {
        ...
        my_tile.sync();
    }
}
```
Cooperative Groups - Particle Simulation

Phase 1: Integration

Phase 2: Collision Detection

Figure 2: Two phases of a particle simulation, with numbered arrows representing the mapping of parallel threads to particles. Note that after integration and construction of the regular grid data structure, the ordering of particles in memory and mapping to threads changes, necessitating a synchronization between phases.
Old Implementation

// threads update particles in parallel
integrate<<<blocks, threads, 0, s>>>(particles);

// Note: implicit sync between kernel launches

// Collide each particle with others in neighborhood
collide<<<blocks, threads, 0, s>>>(particles);
__global__ void particleSim(Particle *p, int N) {

    grid_group g = this_grid();
    // phase 1
    for (i = g.thread_rank(); i < N; i += g.size())
        integrate(p[i]);

    g.sync() // Sync whole grid

    // phase 2
    for (i = g.thread_rank(); i < N; i += g.size())
        collide(p[i], p, N);
}
CUDA 10.0

By Pramod Ramarao
NVIDIA
New Features

• Support for Turing
• CUDA graphs
• New asynchronous task-graph programming model
• New profiler and debugger
# New Turing Warp Matrix Functions

<table>
<thead>
<tr>
<th>Native Types</th>
<th>Input Precision</th>
<th>Output</th>
<th>Supported Sizes</th>
<th>Max Ops/Clock/SM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>half</td>
<td>half or float</td>
<td>16 x 16 x 16</td>
<td>1024</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>integer (int32)</td>
<td>32 x 8 x 16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned char</td>
<td>integer (int32)</td>
<td>8 x 32 x 16</td>
<td></td>
</tr>
<tr>
<td>Experimental</td>
<td>precision::u4 (4-bit unsigned)</td>
<td>integer (int32)</td>
<td>8 x 8 x 32</td>
<td>4096</td>
</tr>
<tr>
<td></td>
<td>precision::s4 (4-bit signed)</td>
<td>integer (int32)</td>
<td>8 x 8 x 128</td>
<td>16384</td>
</tr>
<tr>
<td></td>
<td>precision::b1 (1-bit)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CUDA graphs

// Define graph of work + dependencies

cudaGraphCreate(&graph);
cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// Instantiate graph and apply optimizations

cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times

for(int i=0; i<100; i++)
    cudaGraphLaunch(instance, stream);
OpenMP

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

• OpenMP consists of a set of compiler #pragma 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

```cpp
#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 multiple data, 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);
• 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(".
");
```

- Or, equivalently

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

- 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.

```c
float a[8], b[8];
...
#pragma omp simd
for(int n=0; n<8; ++n) a[n] += b[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];
}

Reduction:
int sum=0;
#pragma omp simd reduction(+:sum)
for(int n=0; n<1000; ++n) sum += table[n];
aligned

```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];
}
```

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

```c
#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.
target, target data

• 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...>>
}
```

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

- Restricts the execution of the associated statement / block to a single thread at 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();
}
...
```

```c
#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

- Variables with static storage duration are shared.
- Dynamically allocated objects are shared.
- Variables with automatic storage duration that are declared in a parallel region are private.
- Variables in heap allocated memory are shared. There can be only one shared heap.
- All variables defined outside a parallel construct become shared when the parallel region is encountered.
- Loop iteration variables are private within their loops. The value of the iteration variable after the loop is the same as if the loop were run sequentially.
- Memory allocated within a parallel loop by the alloca function persists only for the duration of one iteration of that loop, and is private for each thread.
private, firstprivate, shared

```cpp
#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();
}
```c
#define 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 is sequence
  #pragma omp parallel for collapse(2)
  for(int y=0; y<25; ++y)
      for(int x=0; x<80; ++x)
      {
          tick(x, y);
      }
```