

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)

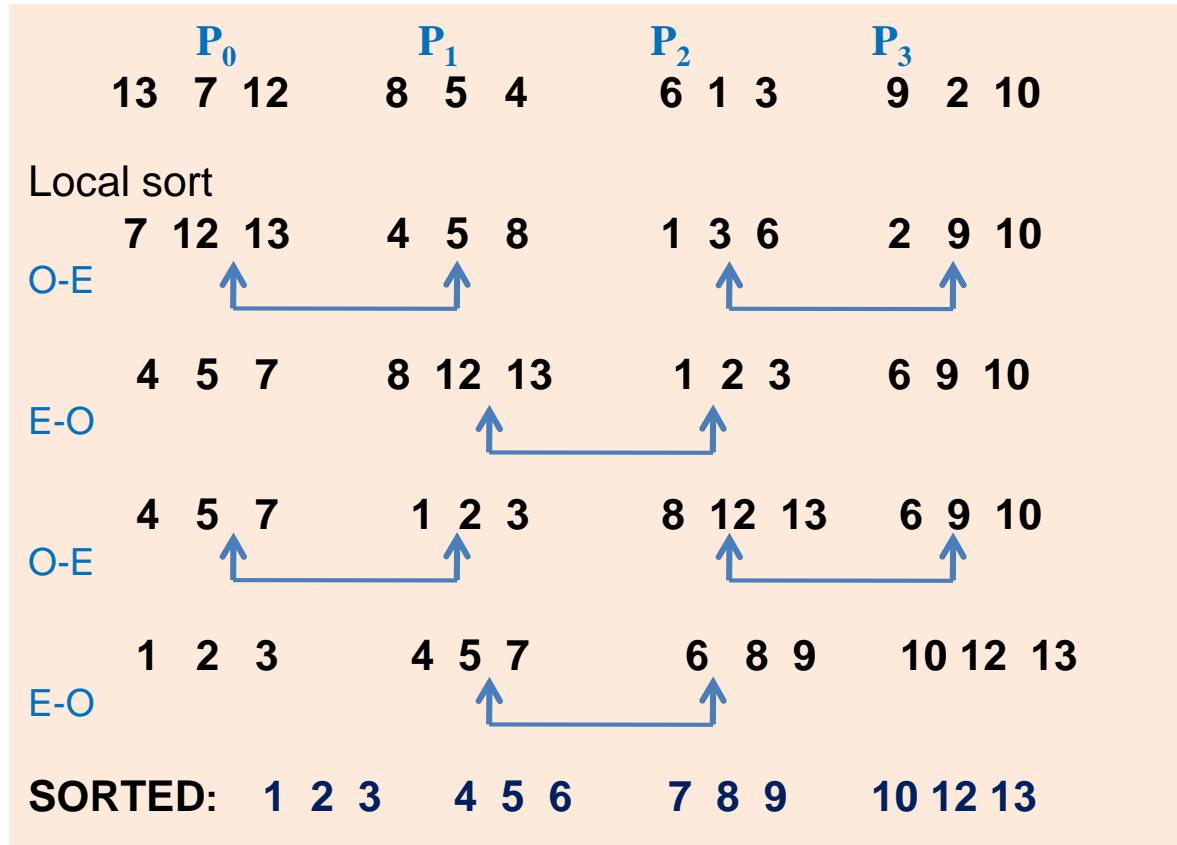
Odd-Even Transposition Sort

Step	P ₀	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇
0	4	2	7	8	5	1	3	6
1	2	4	7	8	1	5	3	6
2	2	4	7	1	8	3	5	6
3	2	4	1	3	7	5	8	6
Time	4	2	1	3	5	7	8	6
5	1	2	3	4	5	7	6	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

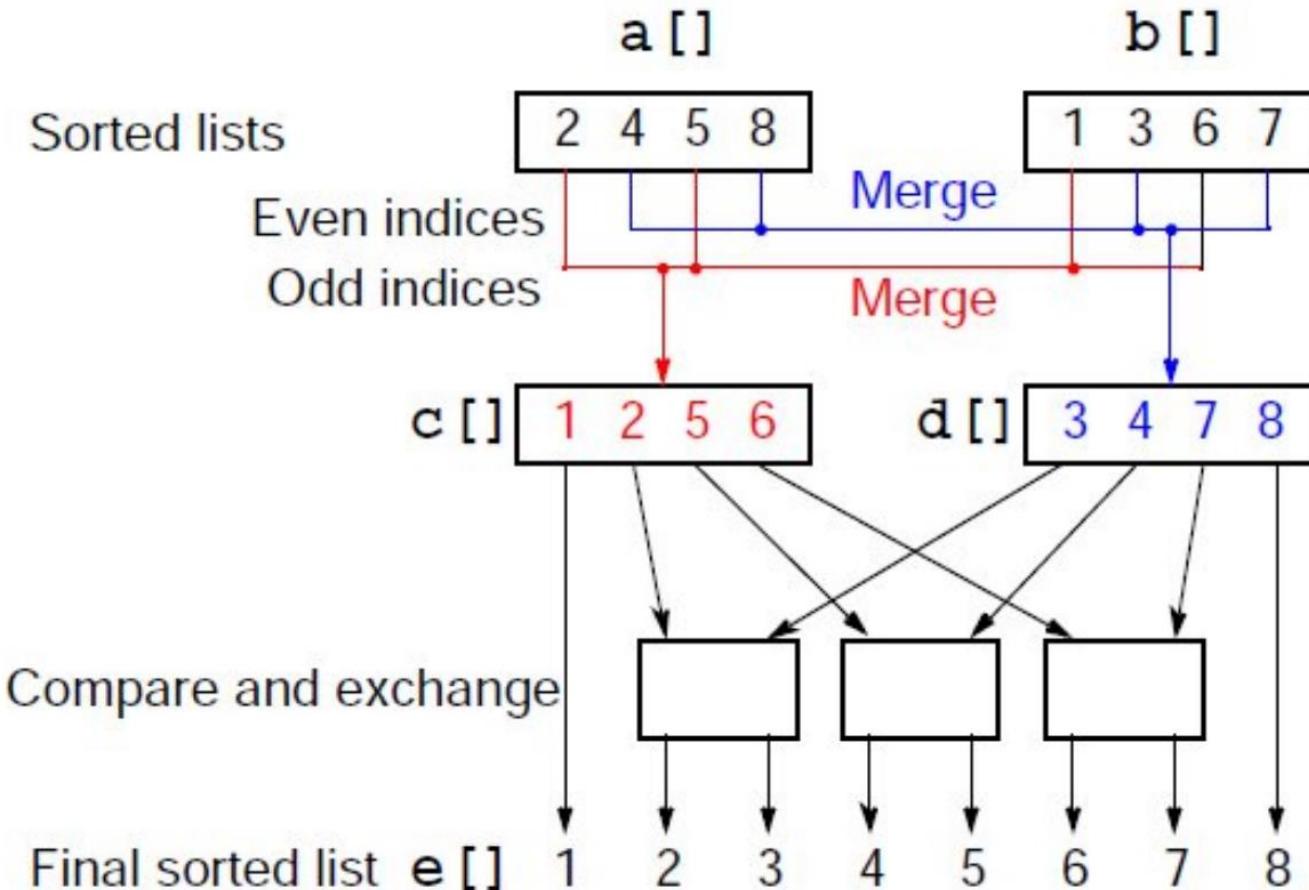
Parallel time complexity: $T_{\text{par}} = O(N)$ (for $P=N$)

Odd-Even Transposition Sort (N>>P)

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

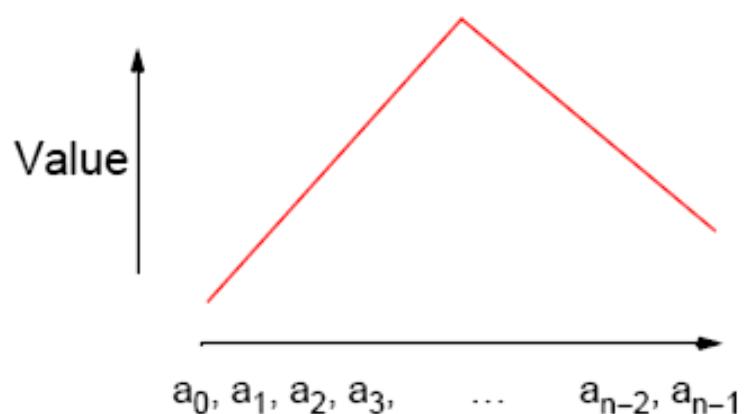


Merge-Split

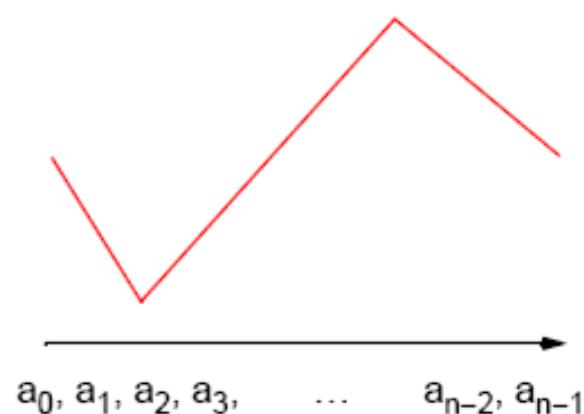


Bitonic Mergesort

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)



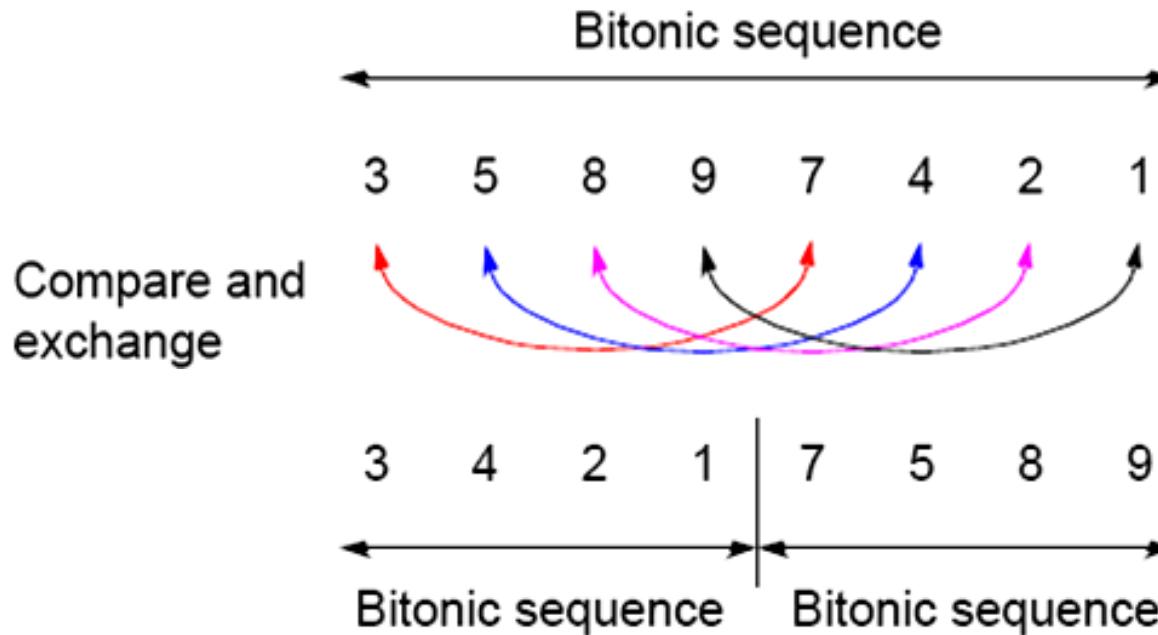
(a) Single maximum



(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 !

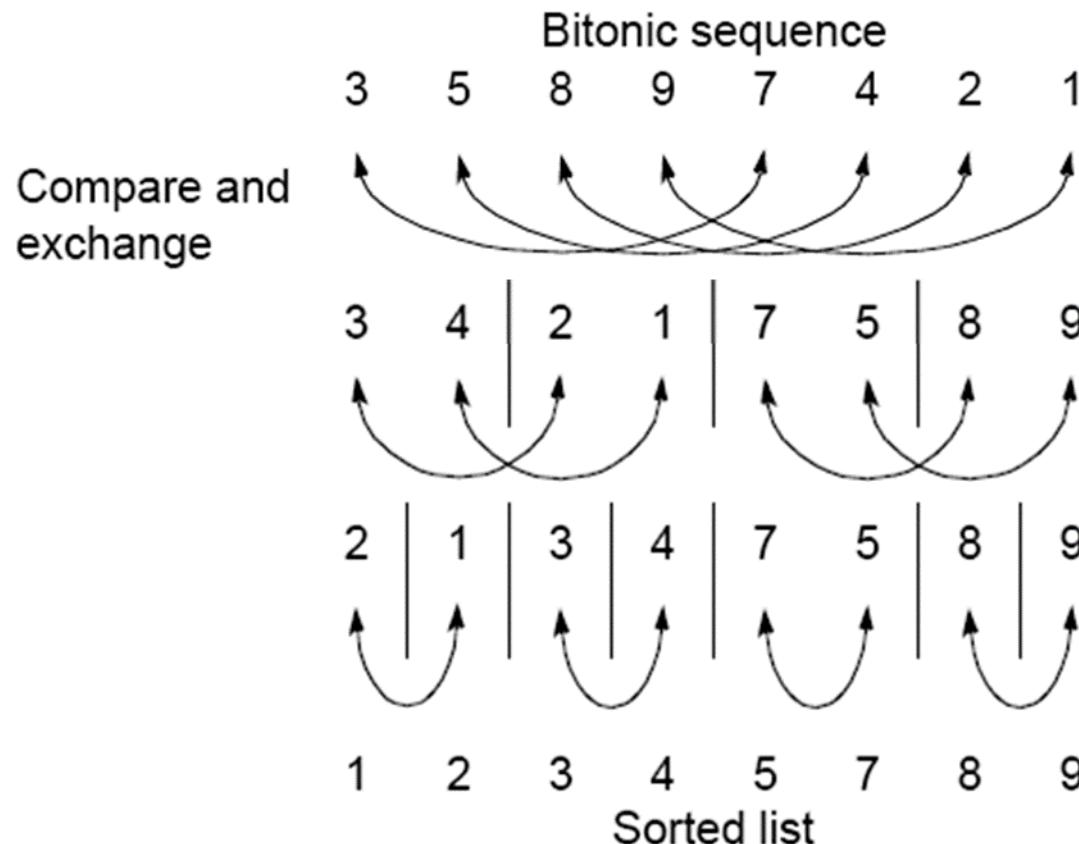
10	11	12	9	.	4	2	5	8		24	20	15	13	.	22	30	32	45					
4	2	.	5	8	10	11	.	12	9		22	20	.	15	13	24	30	.	32	45			
4	.	2	5	.	8	10	.	9	12	.	11	15	.	13	22	.	20	24	.	30	32	.	45
2	4	5	8	9	10	11	12		13	15	20	22	24	30	32	45							

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.



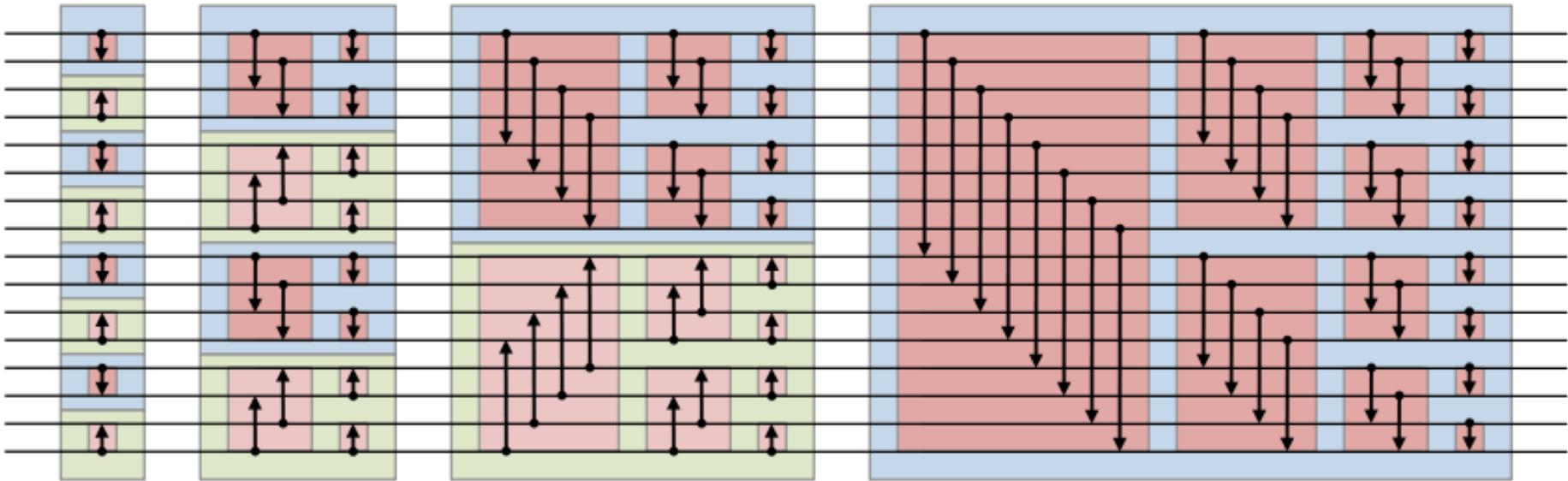
Sorting an Arbitrary Sequence

- 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



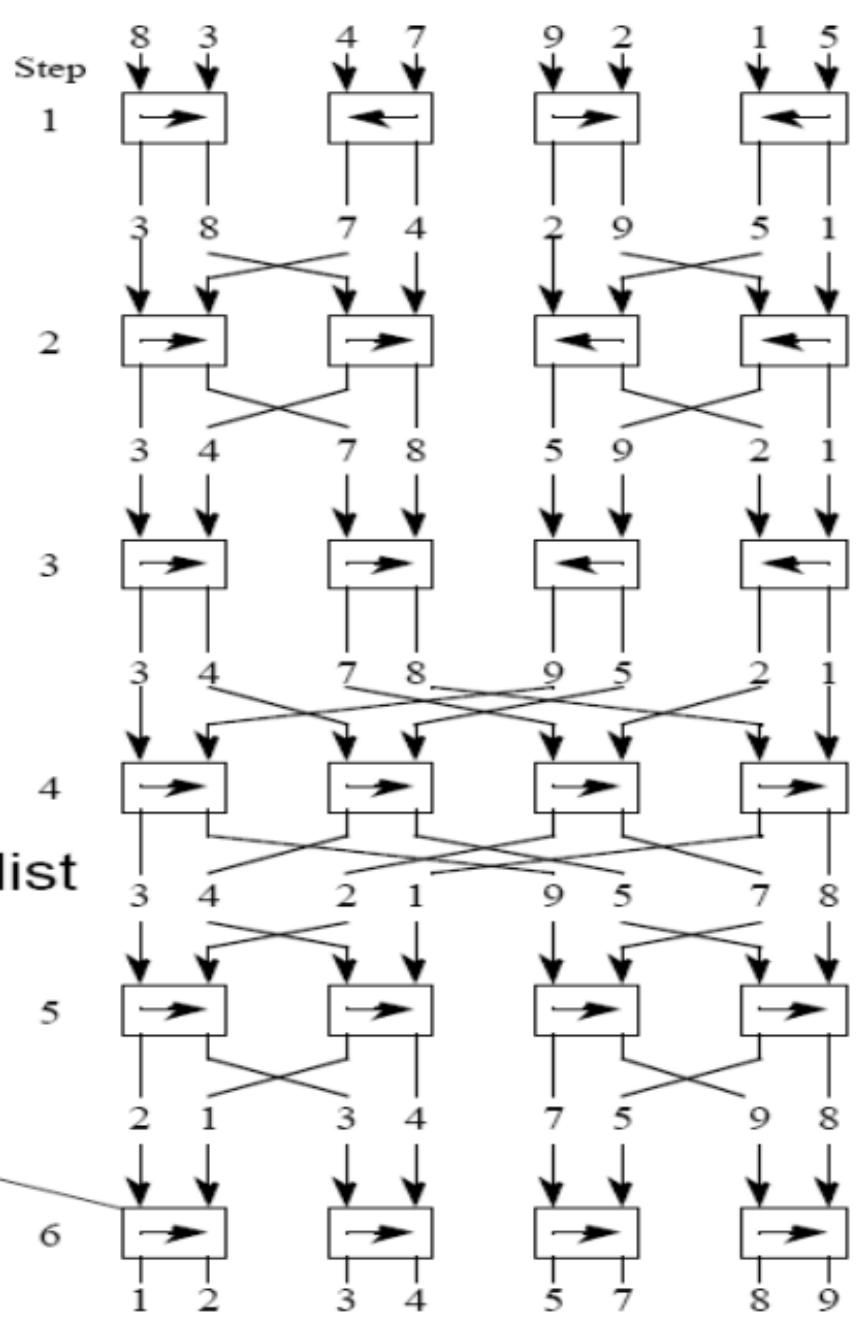
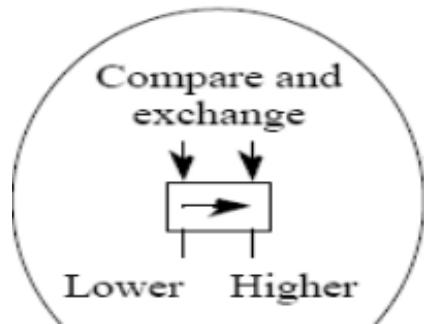
- Whenever two numbers reach the two ends of an arrow, they are compared to ensure that the arrow points toward the larger number.
- If they are out of order, they are swapped.

Source: Wikipedia

Form
bitonic lists
of four
numbers

Form
bitonic list
of eight
numbers

Sort bitonic list



Python Example

```
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)

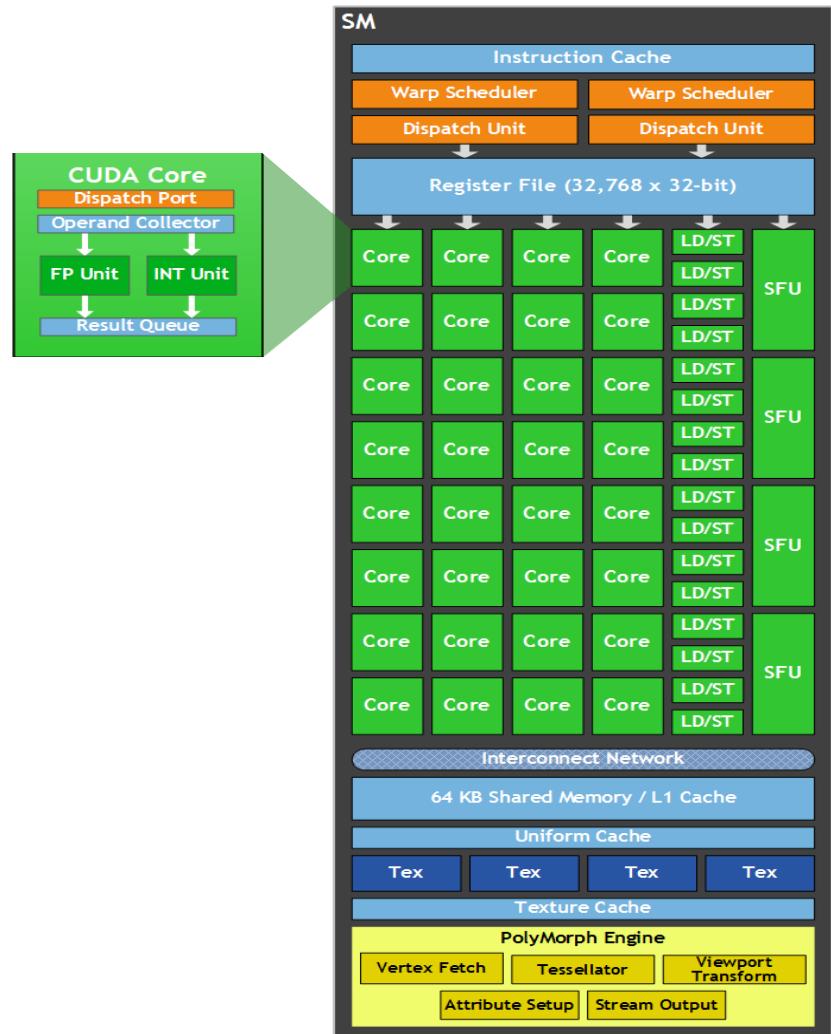
Much Better Compute

- Programmability
 - C++ Support
 - Exceptions/Debug support
- Performance
 - Dual issue SMs
 - L1 cache
 - Larger Shared Memory
 - Much better DP math
 - Much better atomic support
- Reliability: ECC

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

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



Unified Address Space



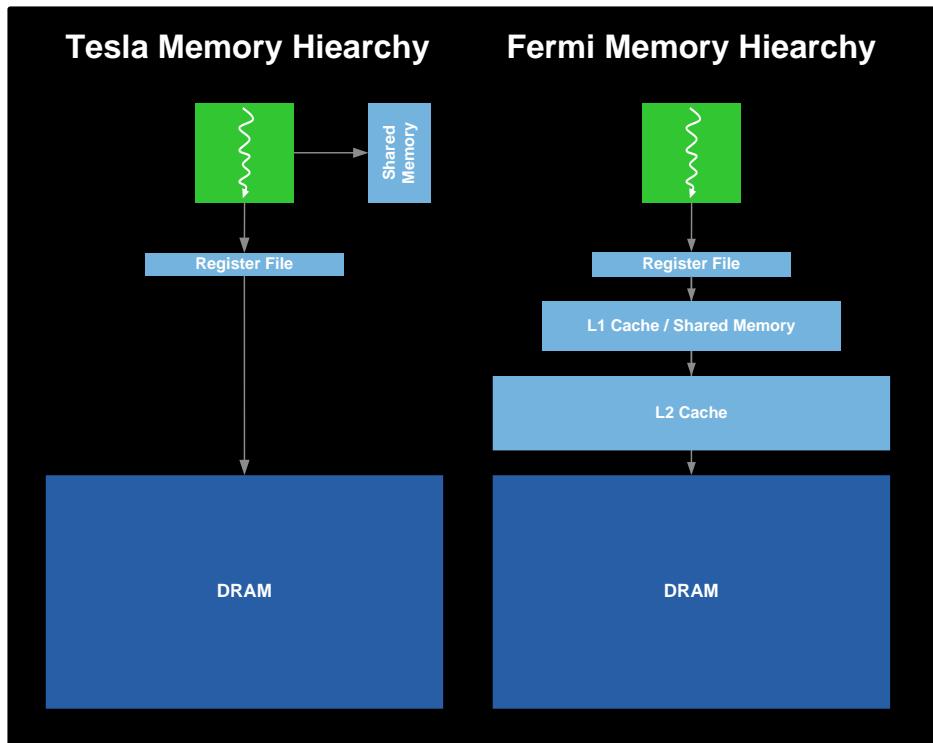
Instruction Issue and Control Flow

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



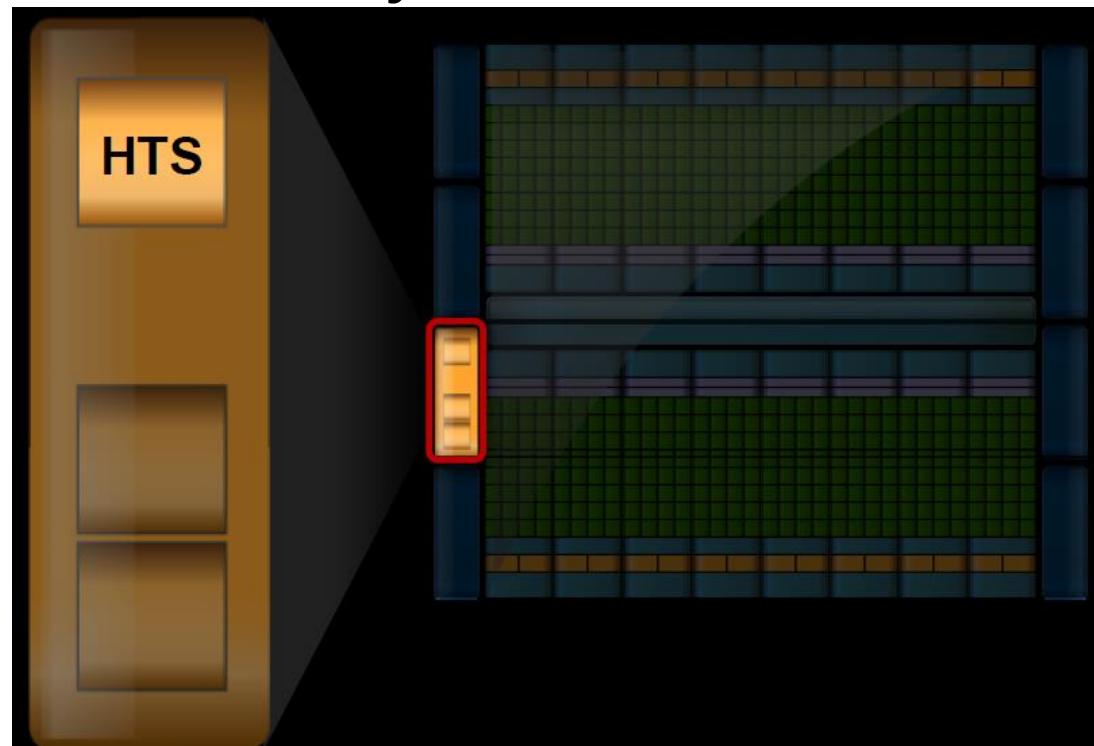
Caches

- 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



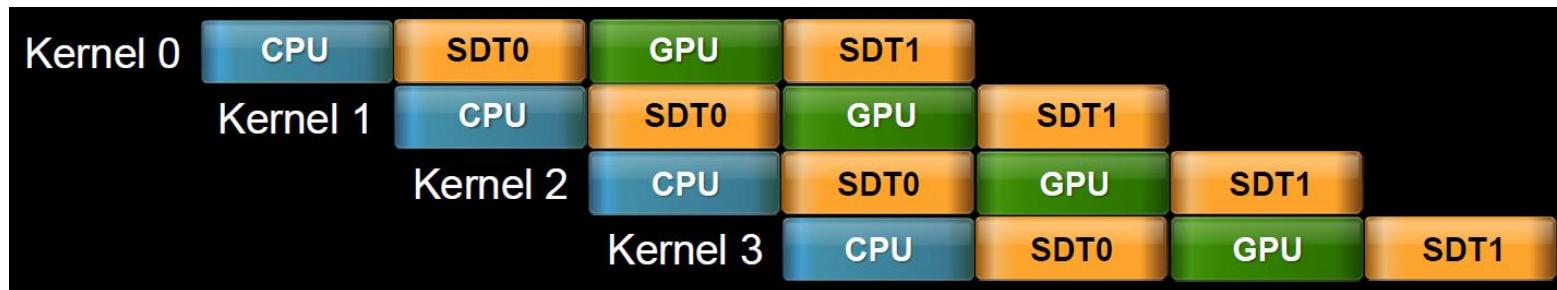
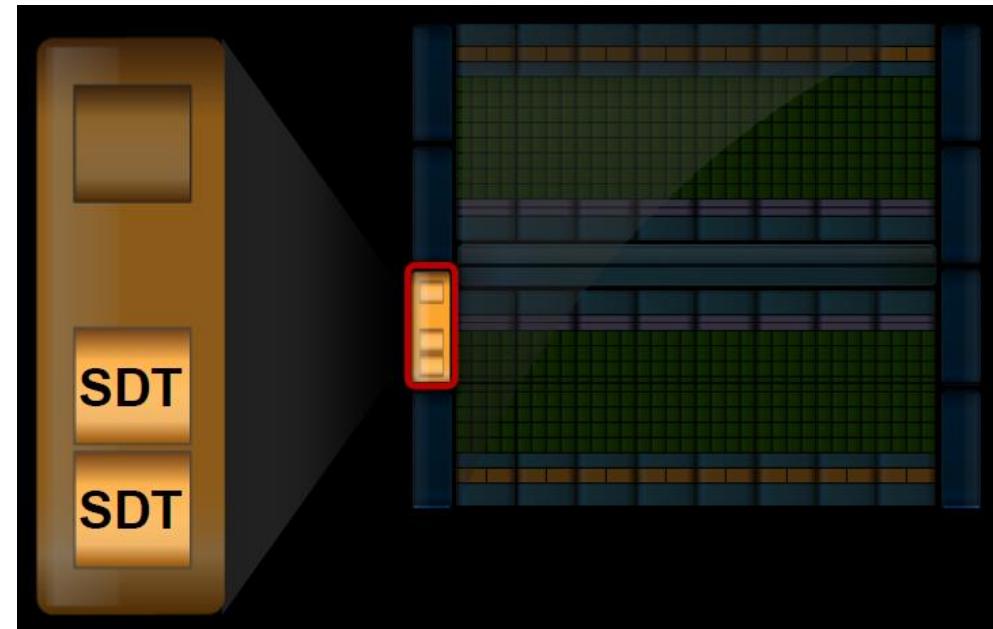
GigaThread Hardware Thread Scheduler

- Hierarchically manages tens of thousands of simultaneously active threads
- 10x faster context switching on Fermi
- Concurrent kernel execution



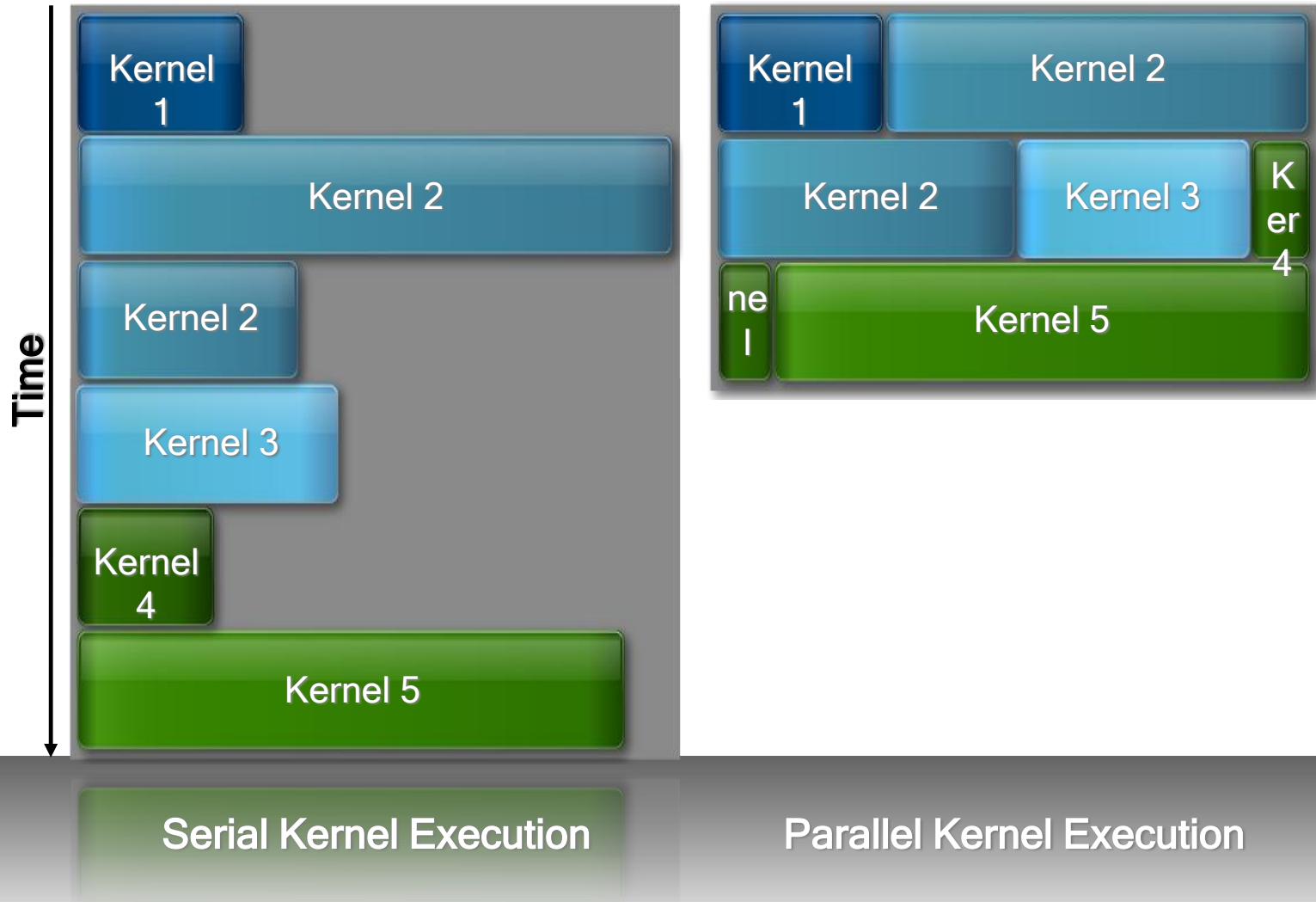
GigaThread Streaming Data Transfer Engine

- 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

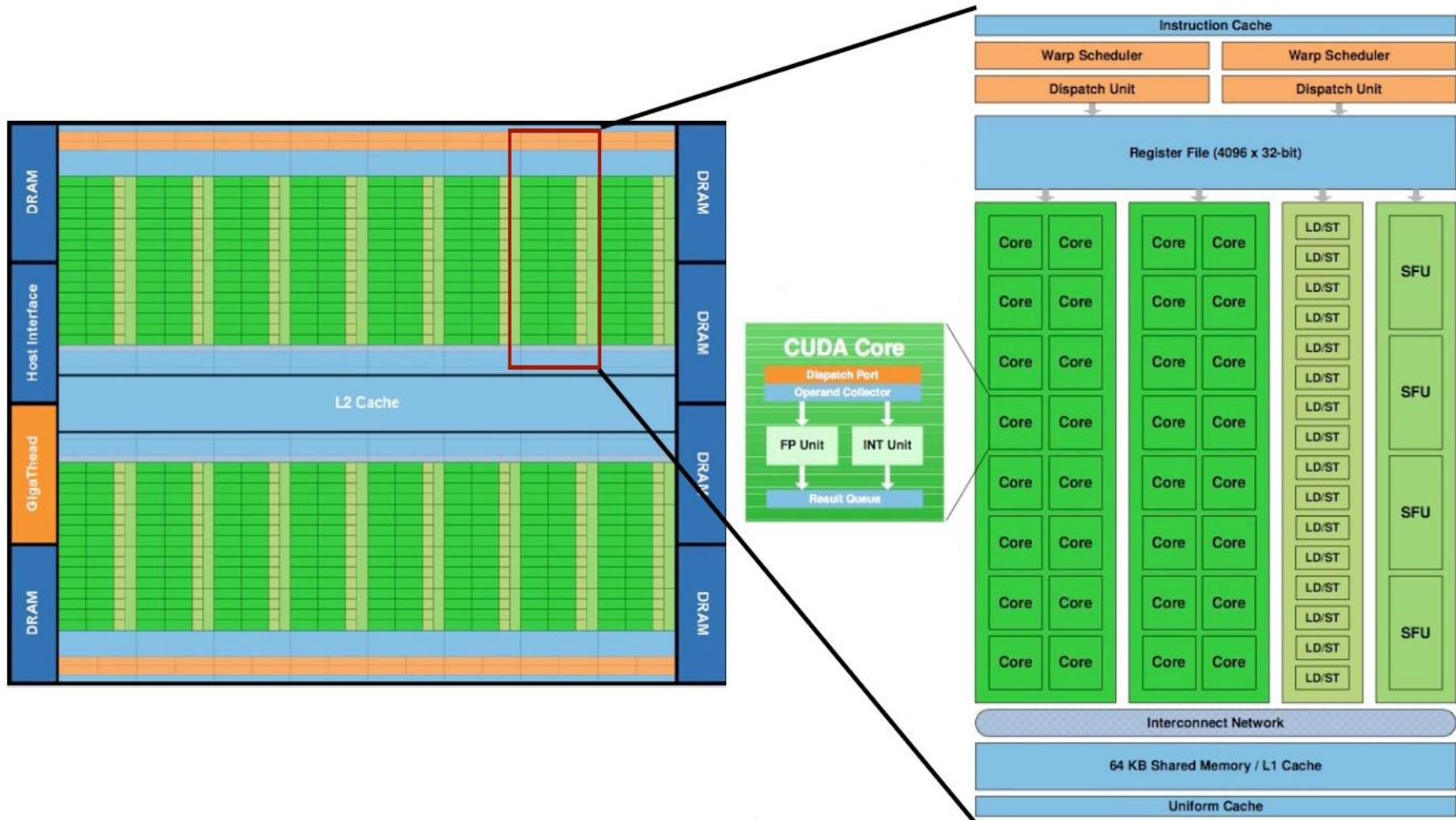
Summary of Features

- 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)

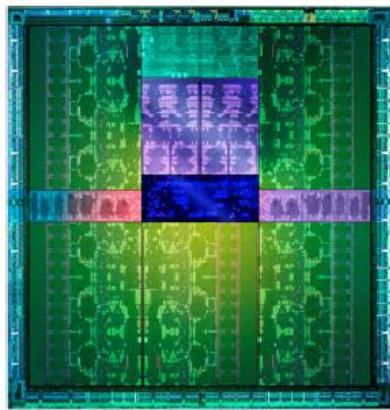
How the Architecture Scales Up

Architecture	G80	GT200	Fermi GF100	Fermi GF104	Kepler GK104	Kepler GK110
Time frame	2006-07	2008-09	2010	2011	2012	2013
CUDA Compute Capability (CCC)	1.0	1.2	2.0	2.1	3.0	3.5
N (multiprocs.)	16	30	16	7	8	15
M (cores/multip.)	8	8	32	48	192	192
Number of cores	128	240	512	336	1536	2880

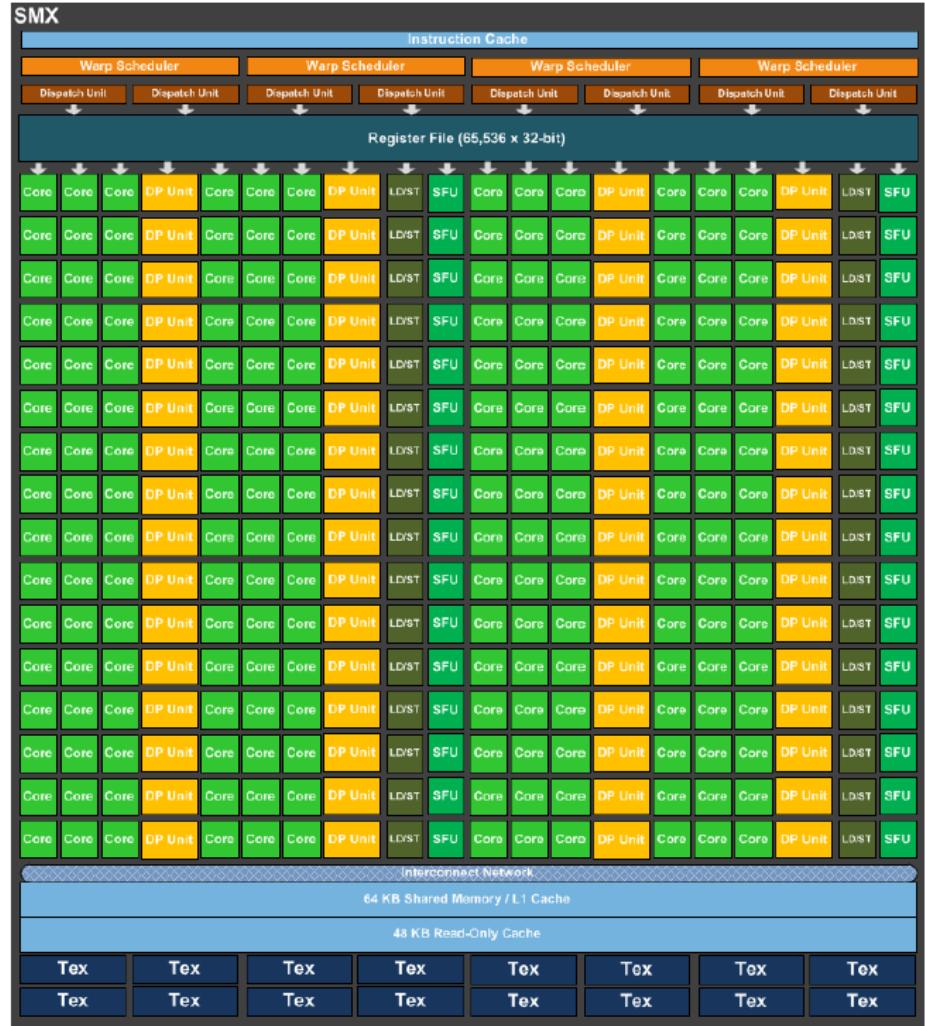
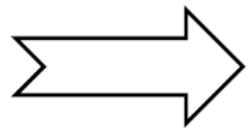
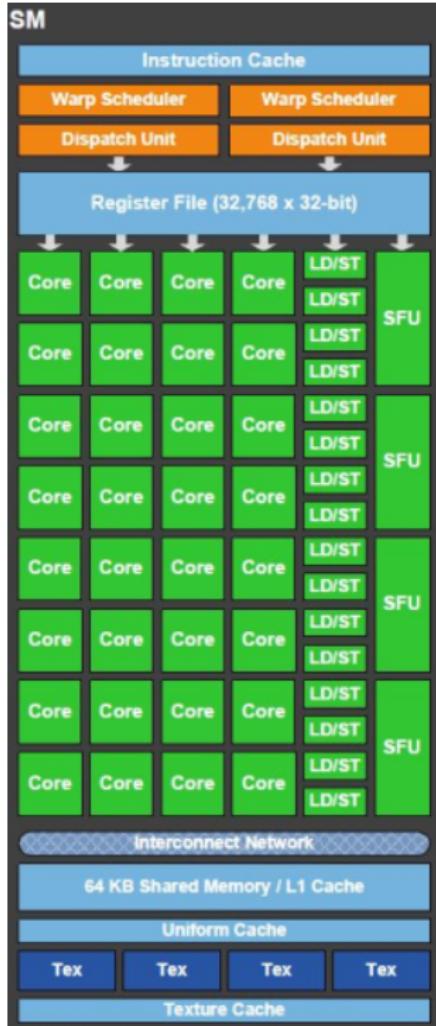
Fermi



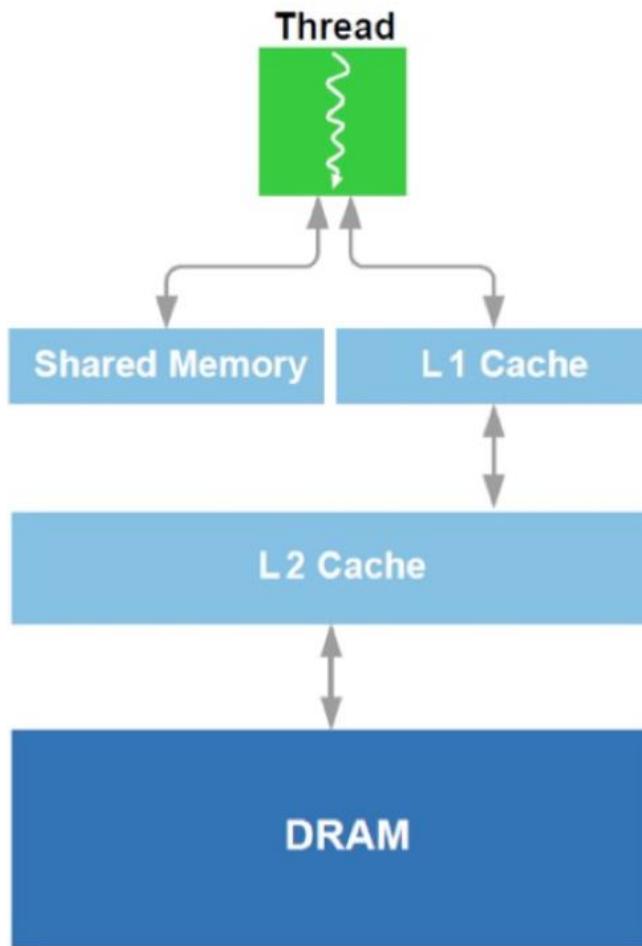
Kepler GK110



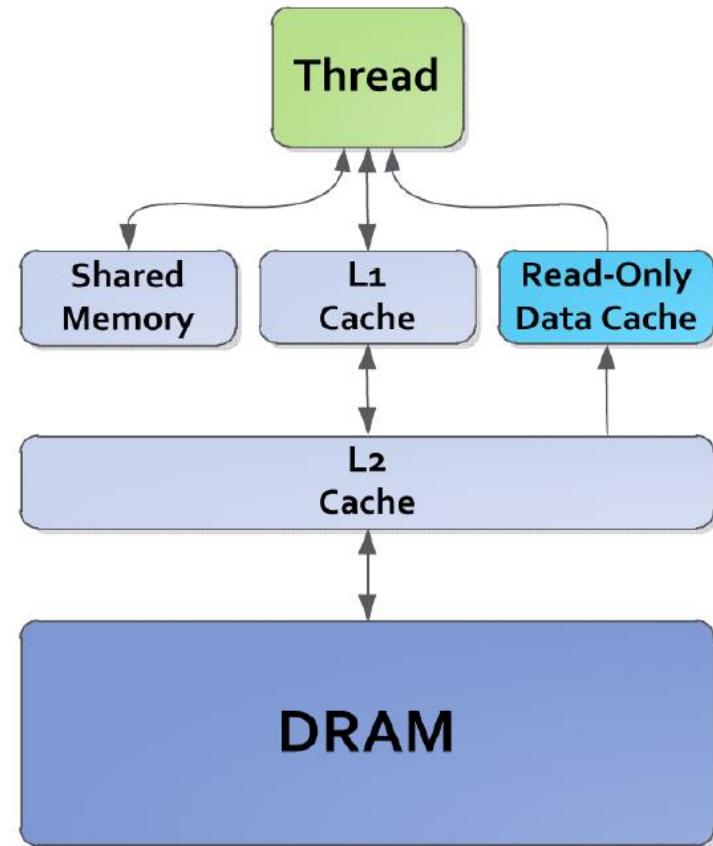
From SM to SMX in Kepler



Differences in Memory Hierarchy



Kepler Memory Hierarchy



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

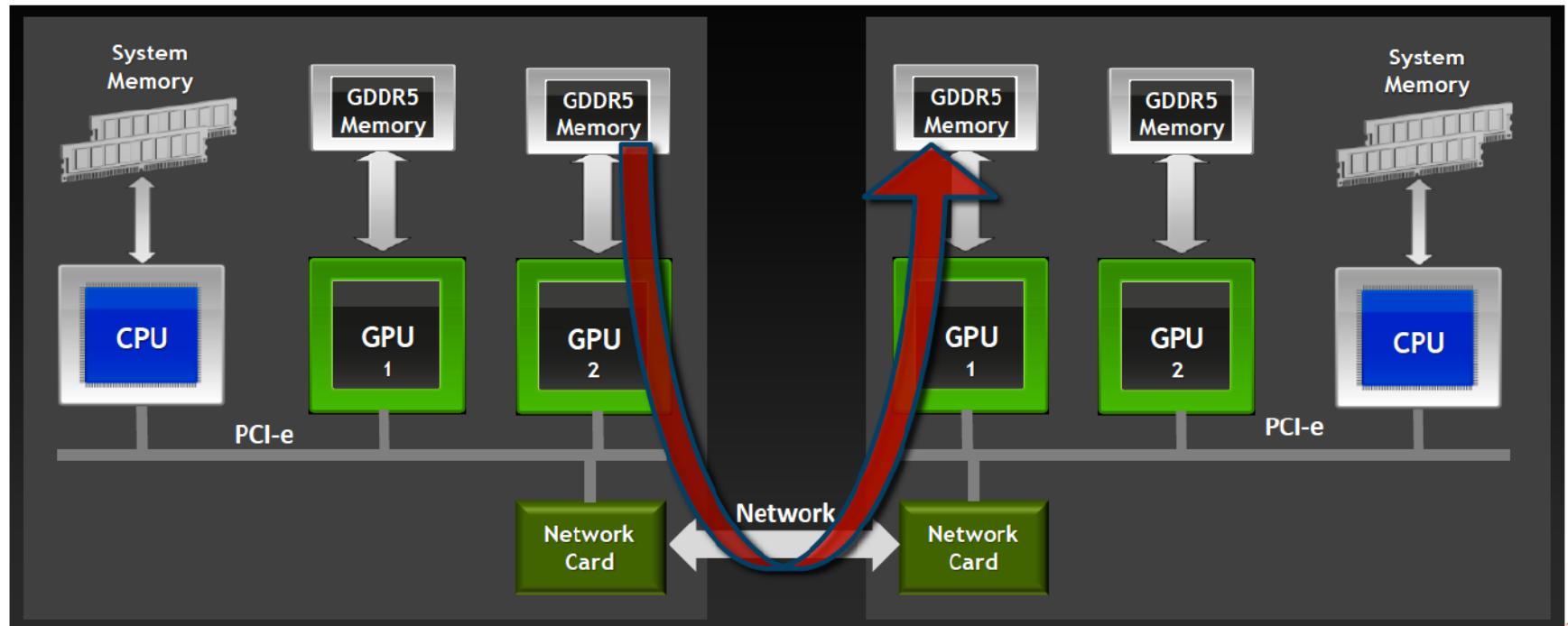
- Annotate eligible kernel parameters with "const __restrict"
- Compiler will automatically map loads to use read-only data cache path.

```
__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

GPU generation	Fermi		Kepler	
Hardware model	GF100	GF104	GK104	GK110
CUDA Compute Capability (CCC)	2.0	2.1	3.0	3.5
Number of threads / warp (warp size)	32	32	32	32
Max. number of warps / Multiprocessor	48	48	64	64
Max. number of blocks / Multiprocessor	8	8	16	16
Max. number of threads / Block	1024	1024	1024	1024
Max. number of threads / Multiprocessor	1536	1536	2048	2048

Crucial enhancement
for Hyper-Q (see later)

Major Hardware Enhancements

- Large scale computations

GPU generation	Fermi		Kepler		Limitation	Impact
Hardware model	GF100	GF104	GK104	GK110		
Compute Capability (CCC)	2.0	2.1	3.0	3.5		
Max. grid size (on X dimension)	2^16-1	2^16-1	2^32-1	2^32-1	Software	Problem size

- New architectural features

GPU generation	Fermi		Kepler		Limitation	Impact
Hardware model	GF100	GF104	GK104	GK110		
Compute Capability (CCC)	2.0	2.1	3.0	3.5		
Dynamic Parallelism	No	No	No	Yes	Hardware	Problem structure
Hyper-Q	No	No	No	Yes	Hardware	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



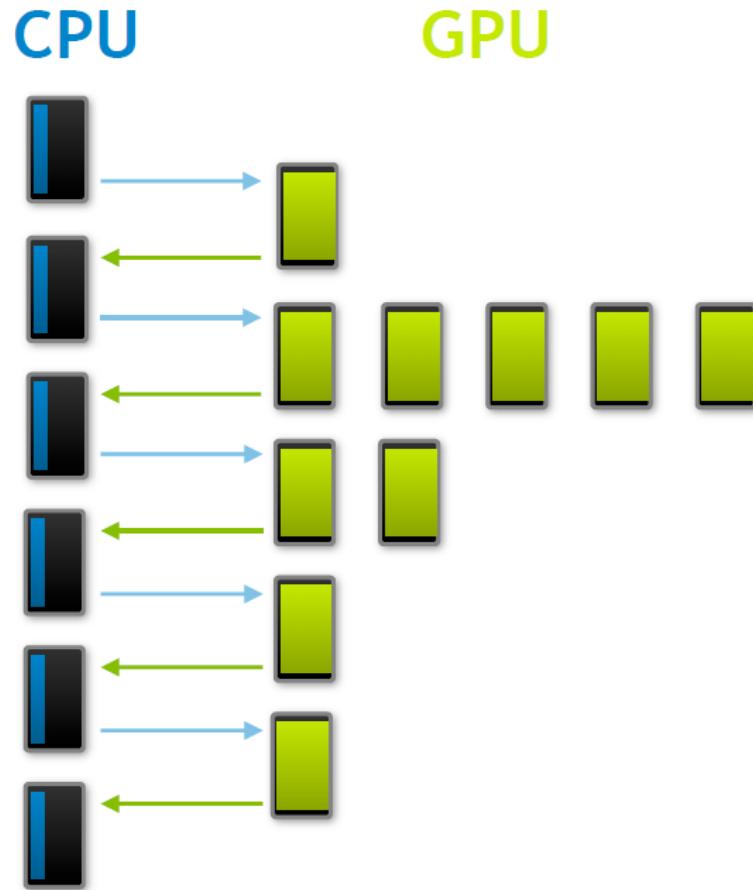
Fermi: Only CPU
can generate GPU work.



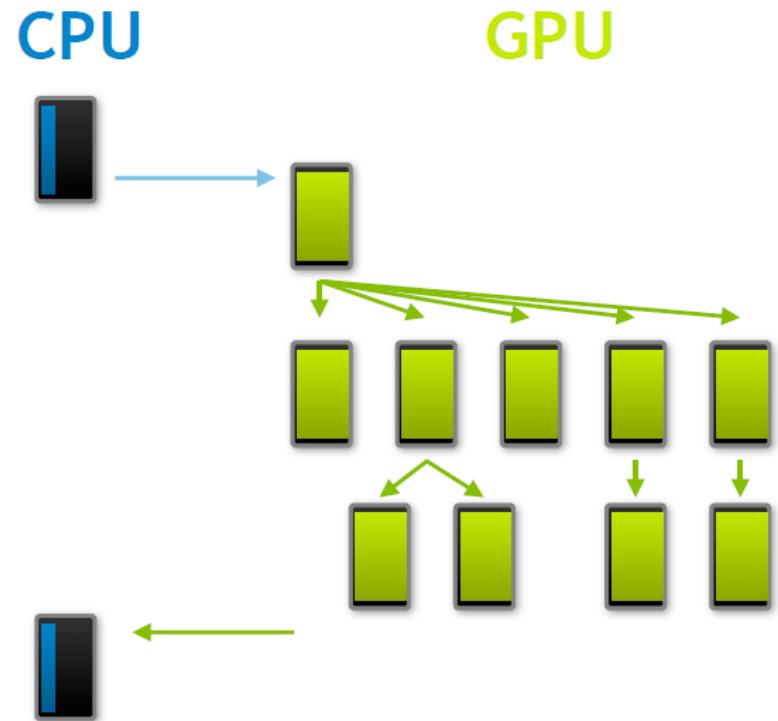
Kepler: GPU can
generate work for itself.

Dynamic Parallelism

The pre-Kepler GPU is a co-processor



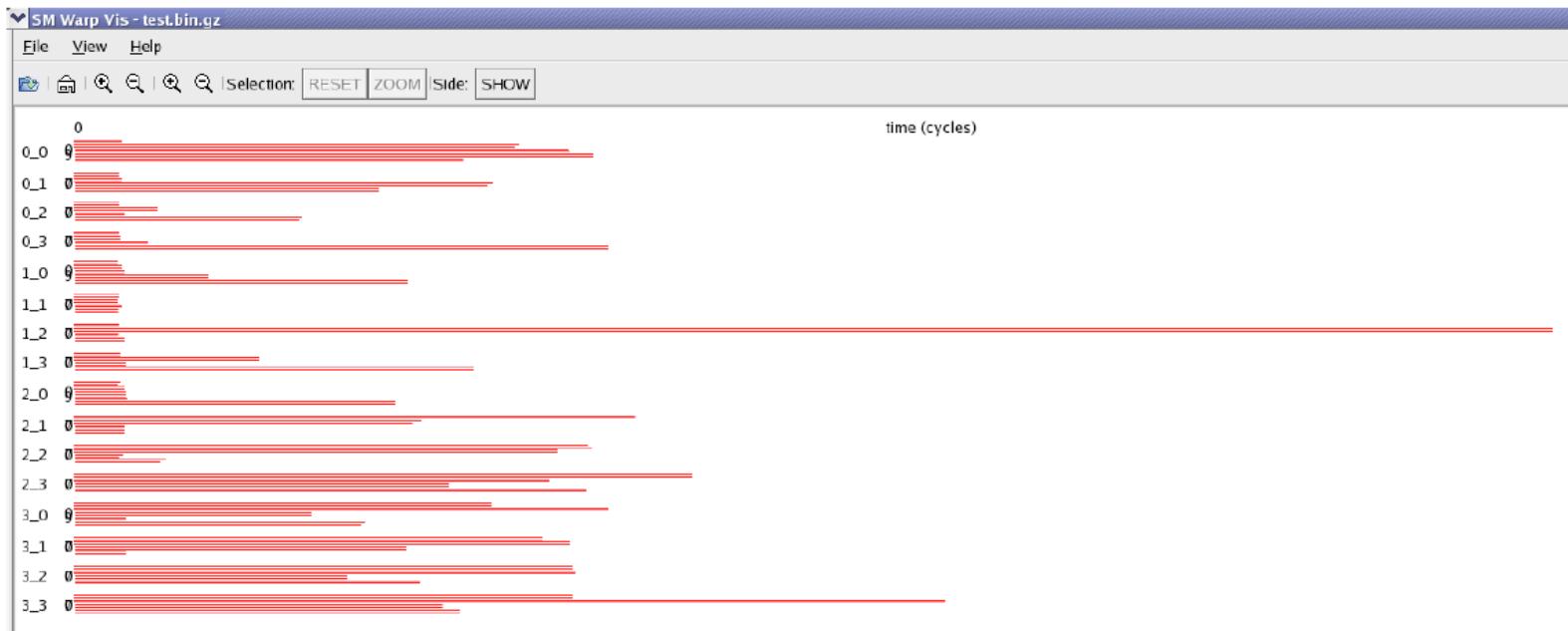
The Kepler GPU is autonomous:
Dynamic parallelism



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

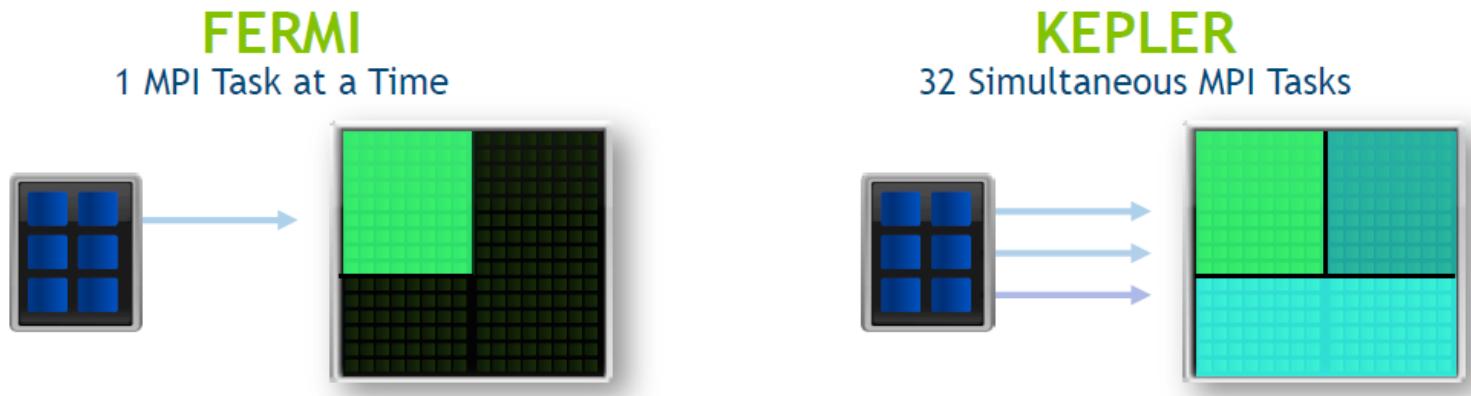
Workload Balance

- 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:

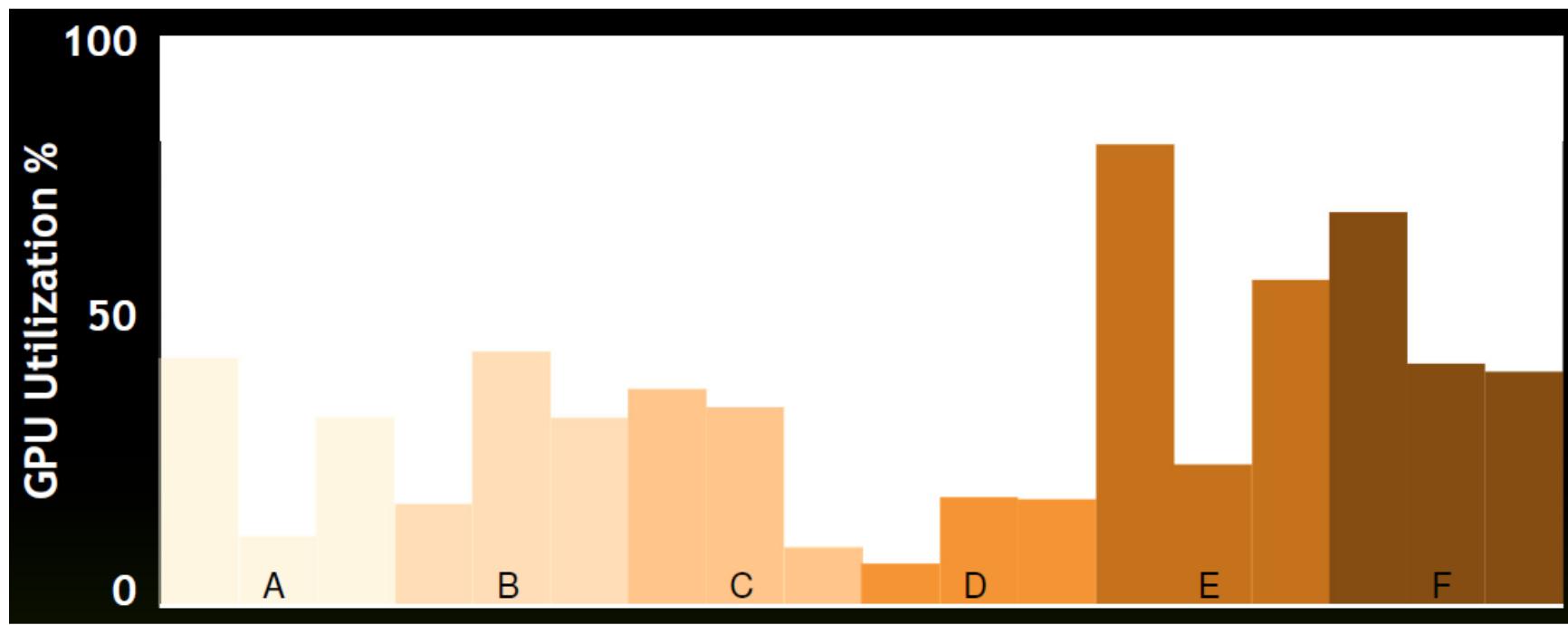


Hyper-Q

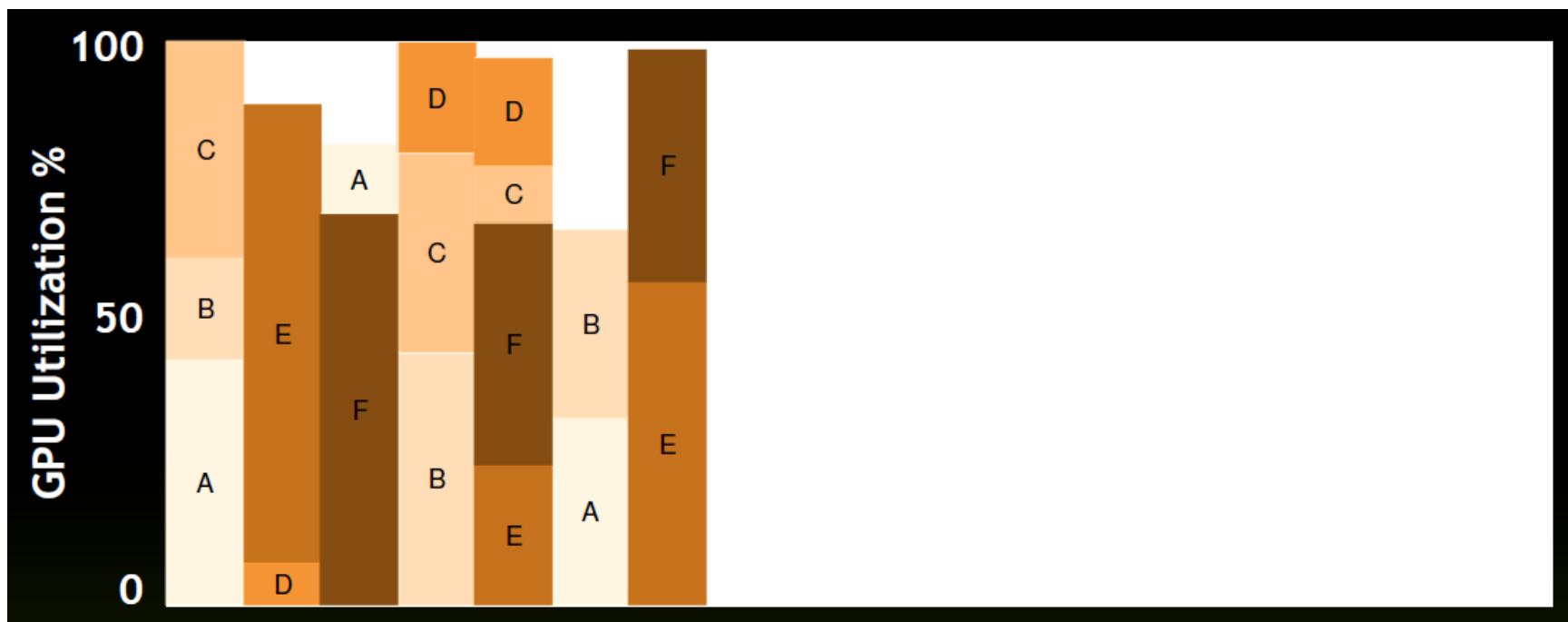
- In Fermi, several CPU processes can send thread blocks to the same GPU, but a kernel cannot start its execution until the previous one has finished
- In Kepler, we can execute simultaneously up to 32 kernels launched from different:
 - MPI processes, CPU threads (POSIX threads) or CUDA streams
- This increments the % of temporal occupancy on the GPU



Without Hyper-Q

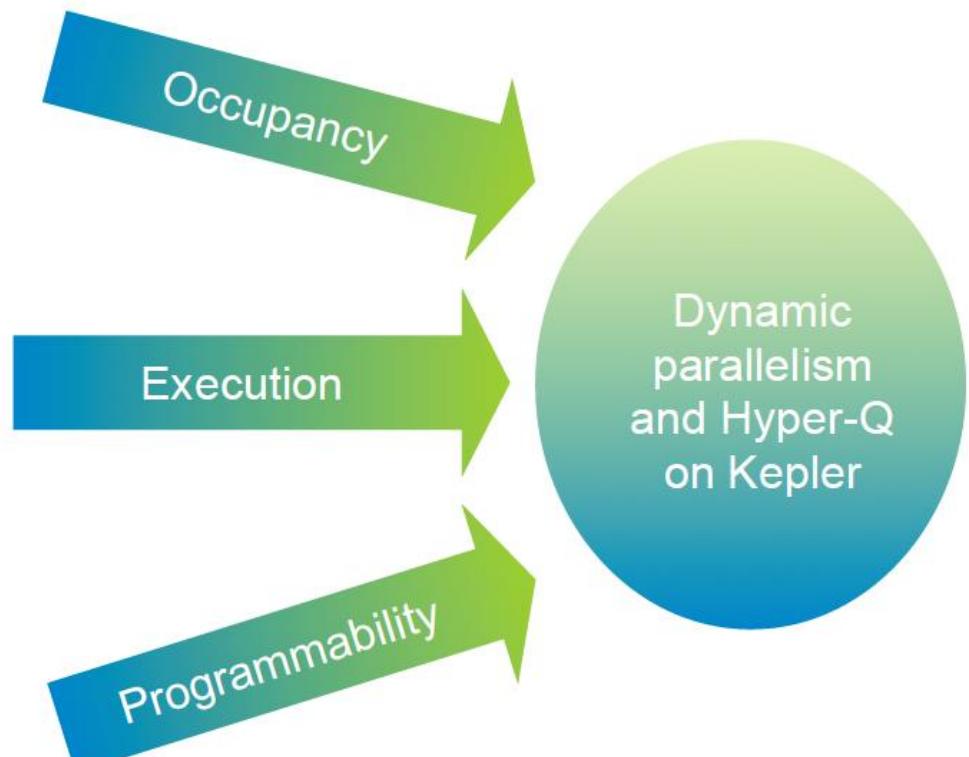


With Hyper-Q



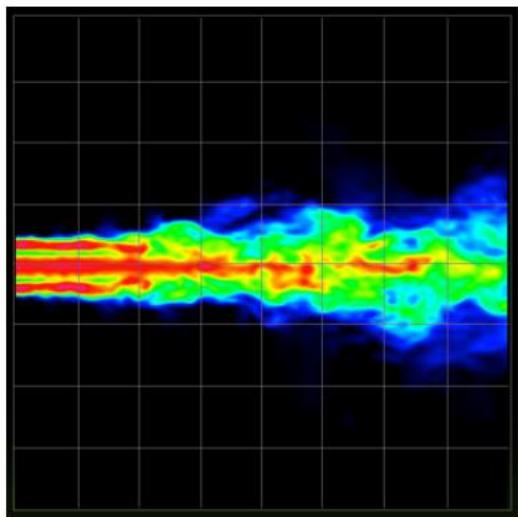
Six Ways to Improve Code on Kepler

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



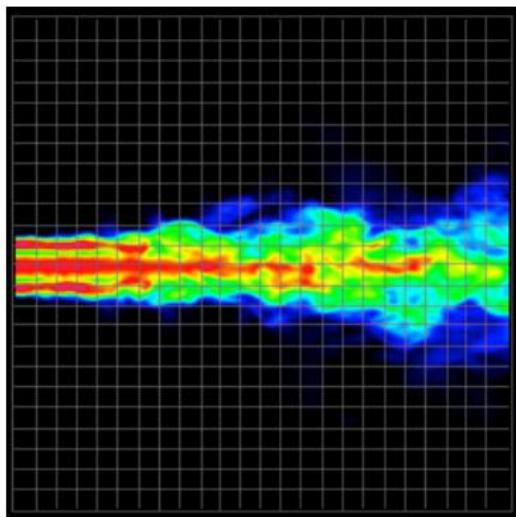
Dynamic Work Generation

Coarse grid



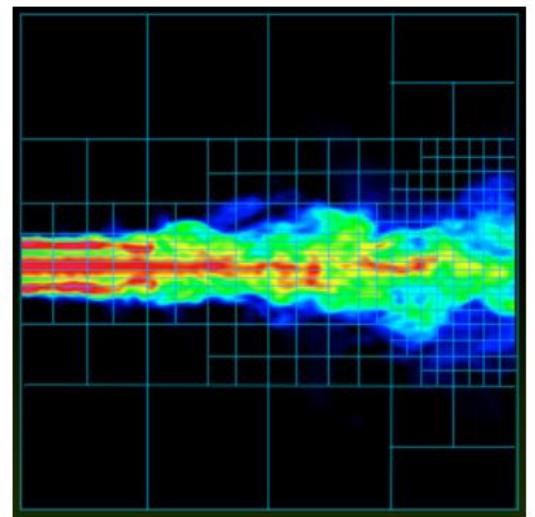
Higher performance,
lower accuracy

Fine grid



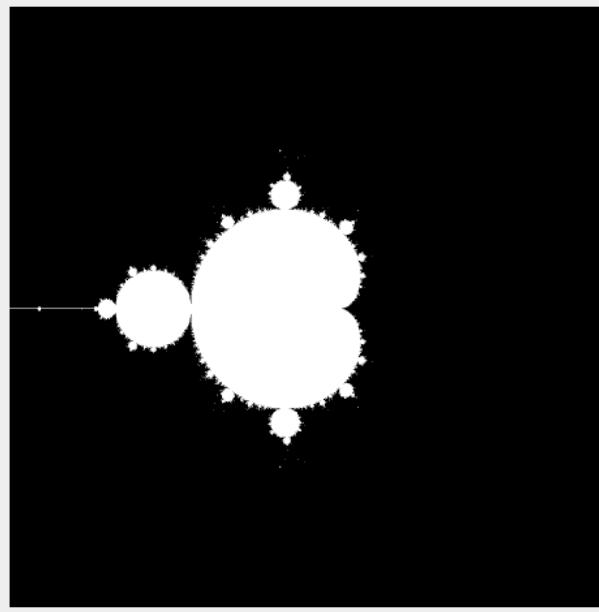
Lower performance,
higher accuracy

Dynamic grid

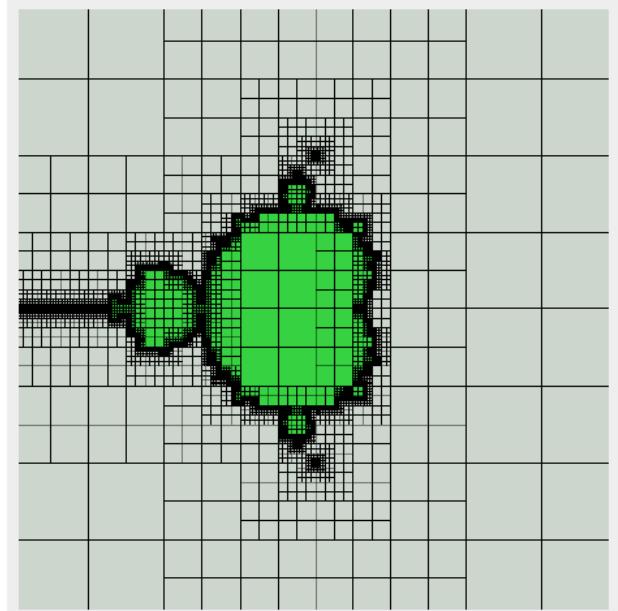


Target performance
where accuracy is required

Parallelism based on Level of Detail



Computational power allocated to regions of interest



CUDA until 2012:

- The CPU launches kernels regularly.
- All pixels are treated the same.

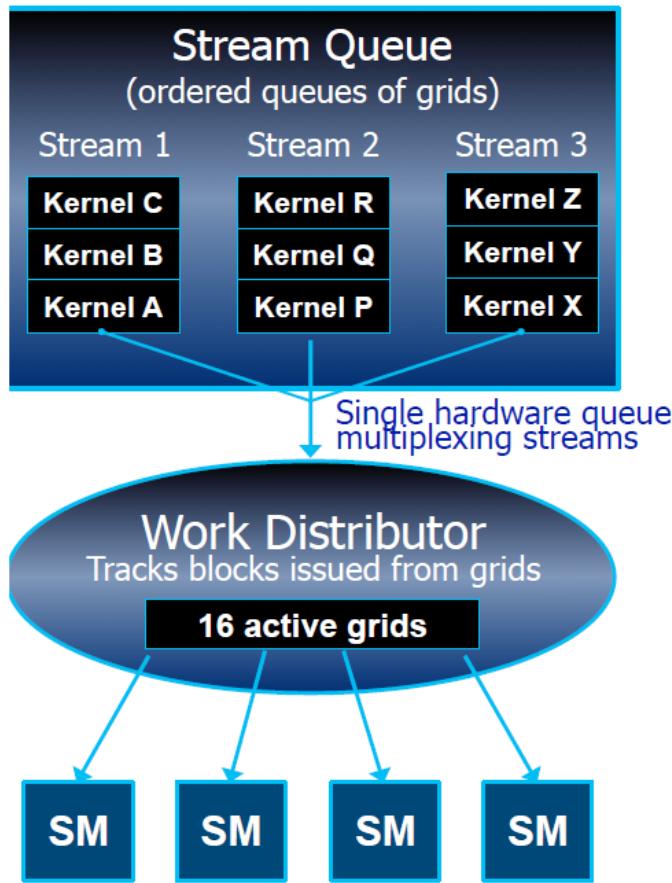


CUDA on Kepler:

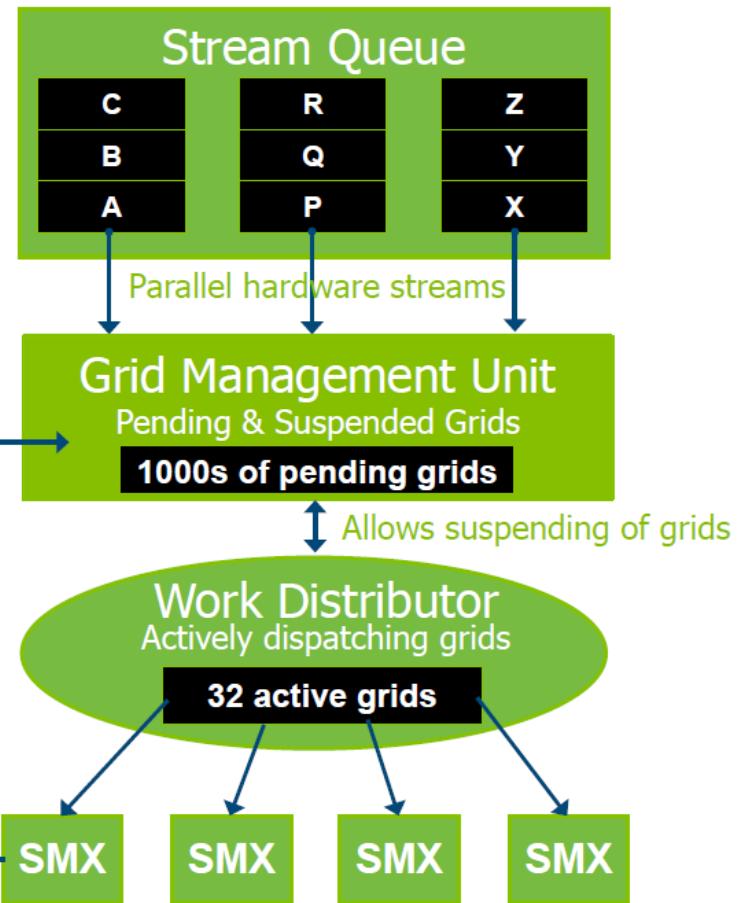
- The GPU launches a different number of kernels/blocks for each computational region.

Grid Management Unit

Fermi



Kepler GK110



Software and Hardware Queues

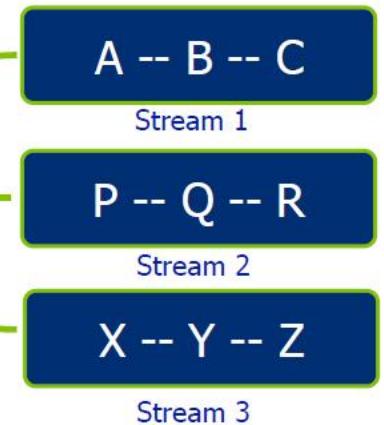
Fermi:

Up to 16 grids
can run at once
on GPU hardware

But CUDA streams multiplex into a single queue

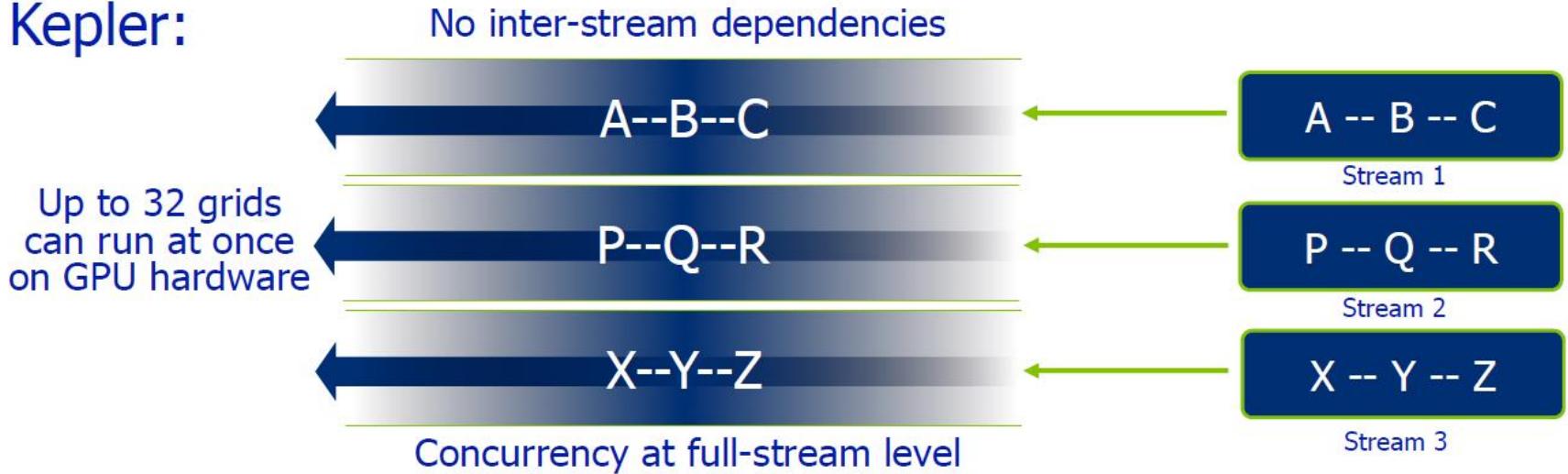


Chances for overlapping: Only at stream edges



Software and Hardware Queues

Kepler:



Instruction Issue and Execution

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

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:

```
__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);
```

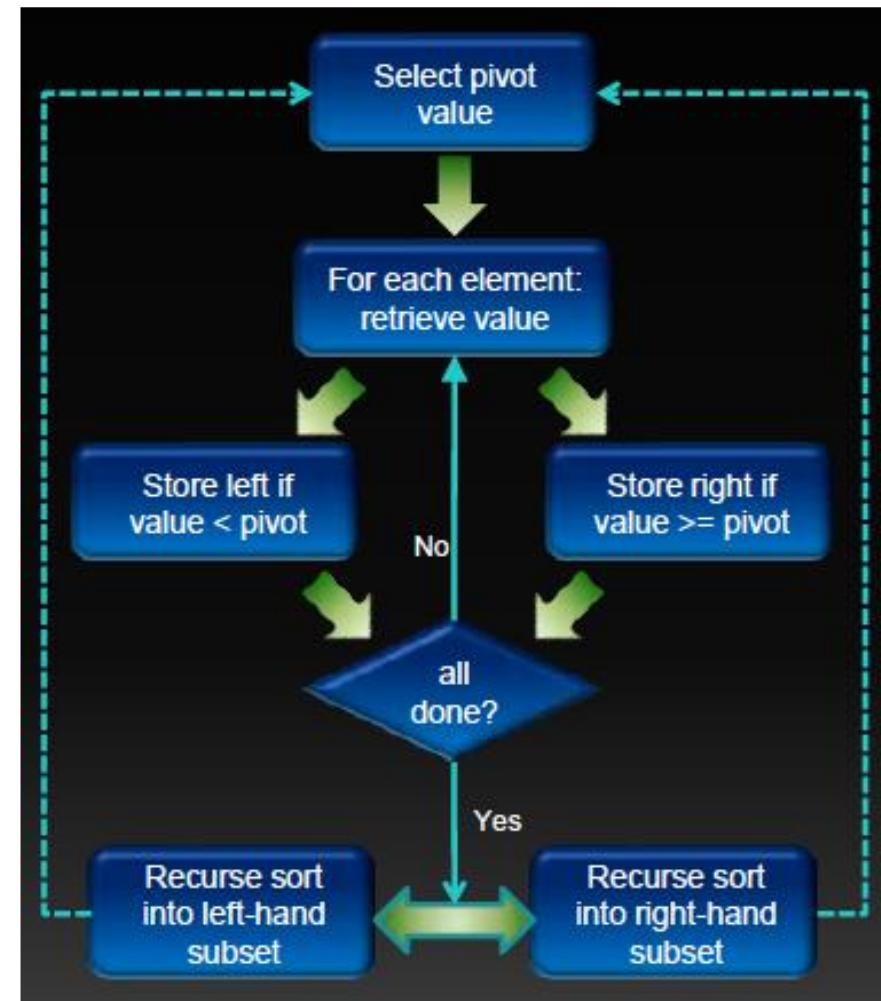
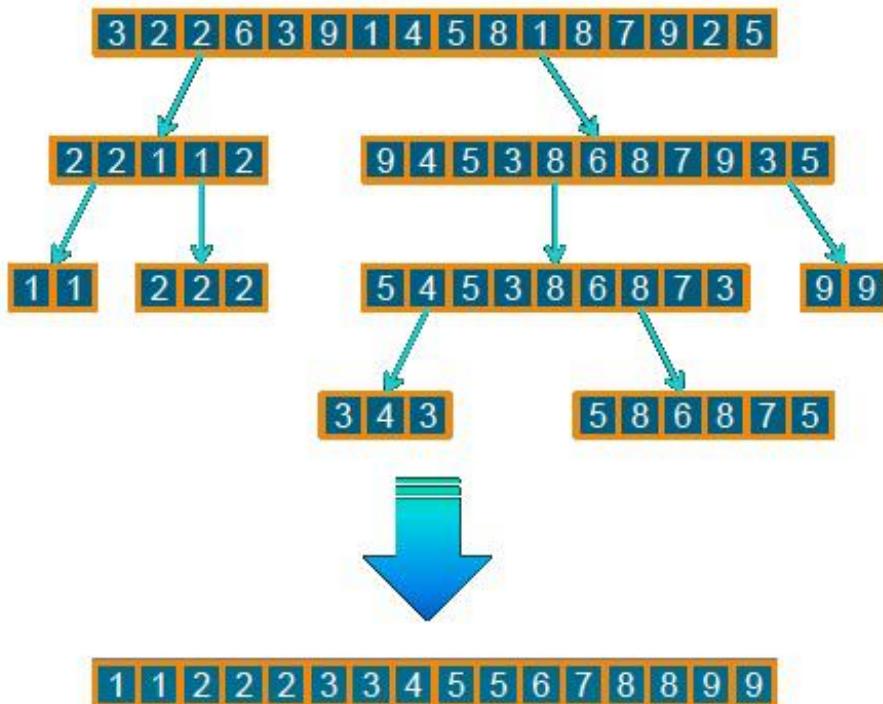
- Up to 24 nested loops supported in CUDA 5.0

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

```
_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; lx = lptr-data;
    if (l < rx)
        qsort<<<...>>>(data,l,rx);
    if (r > lx)
        qsort<<<...>>>(data,lx,r);
}
```

left- and right-hand sorts are serialized

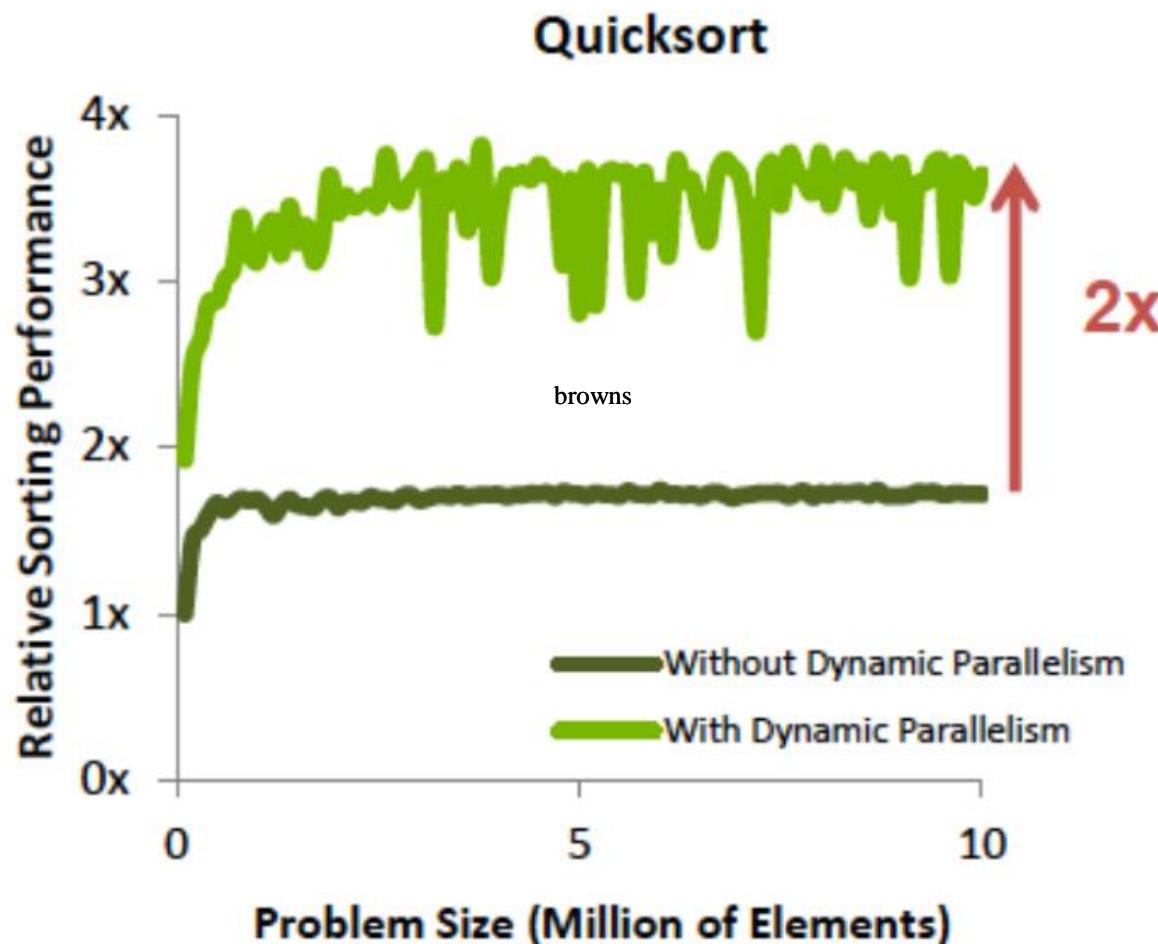
Version for Kepler

```
_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);

    // Use streams this time for the recursion
    cudaStream_t s1, s2;
    cudaStreamCreateWithFlags(&s1, ...);
    cudaStreamCreateWithFlags(&s2, ...);
    int rx = rptr-data; lx = lptr-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

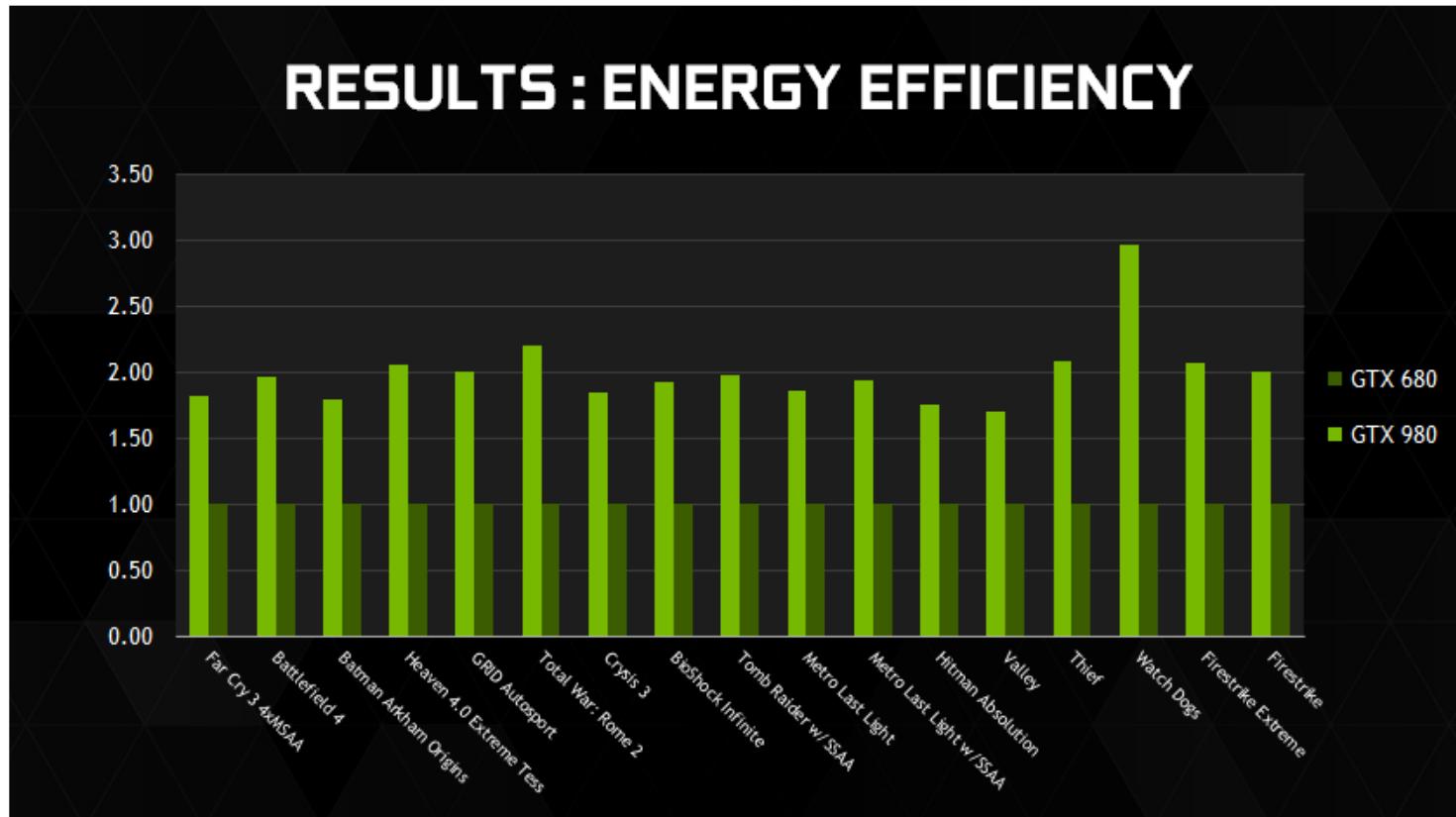


Maxwell (2nd generation)

Released in 2014

Material by Mark Harris (NVIDIA)
and others

Energy Efficiency



Performance per Watt

GTX 680: Kepler GTX 980: Maxwell

New Features

GPU	GeForce GTX 680 (Kepler)	GeForce GTX 980 (Maxwell)
SMs	8	16
CUDA Cores	1536	2048
Base Clock	1006 MHz	1126 MHz
GPU Boost Clock	1058 MHz	1216 MHz
GFLOPs	3090	4612 ¹
Texture Units	128	128
Texel fill-rate	128.8 Gigatexels/sec	144.1 Gigatexels/sec
Memory Clock	6000 MHz	7000 MHz
Memory Bandwidth	192 GB/sec	224 GB/sec
ROPs	32	64
L2 Cache Size	512KB	2048KB
TDP	195 Watts	165 Watts
Transistors	3.54 billion	5.2 billion
Die Size	294 mm ²	398 mm ²
Manufacturing Process	28-nm	28-nm

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

Graphics

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

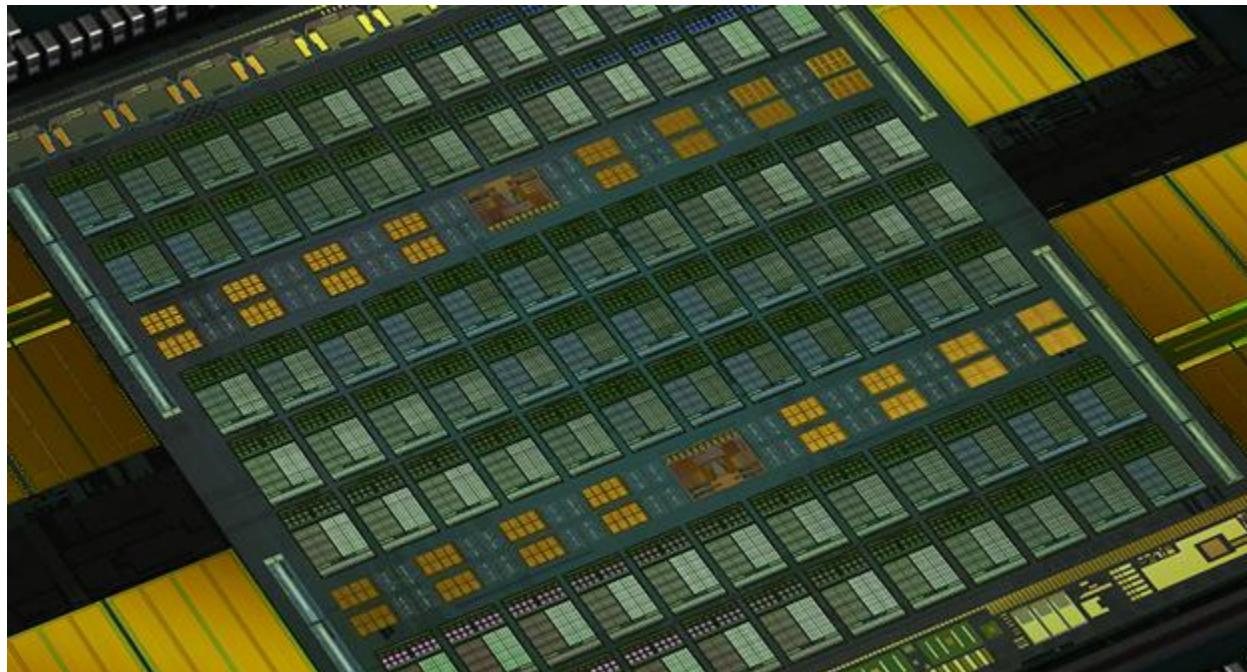


Released in 2016

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

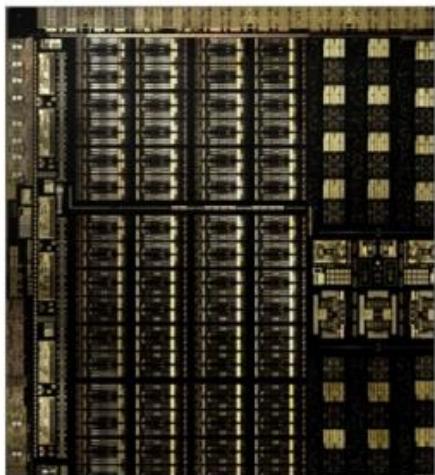
Key New Features

- 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

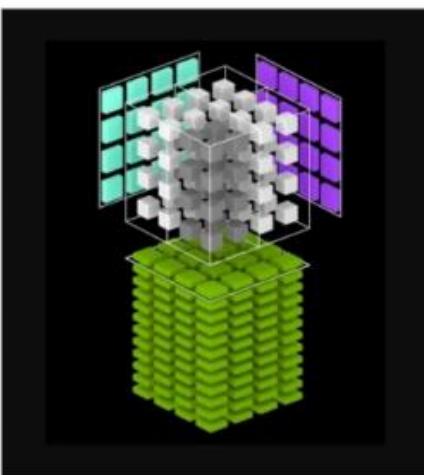
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS ¹	5	6.8	10.6	15.7
Peak FP64 TFLOPS ¹	1.7	.21	5.3	7.8
Peak Tensor TFLOPS ¹	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²	815 mm ²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

¹ Peak TFLOPS rates are based on GPU Boost Clock

Turing



NEW CORE ARCHITECTURE



TENSOR CORE



RT CORE



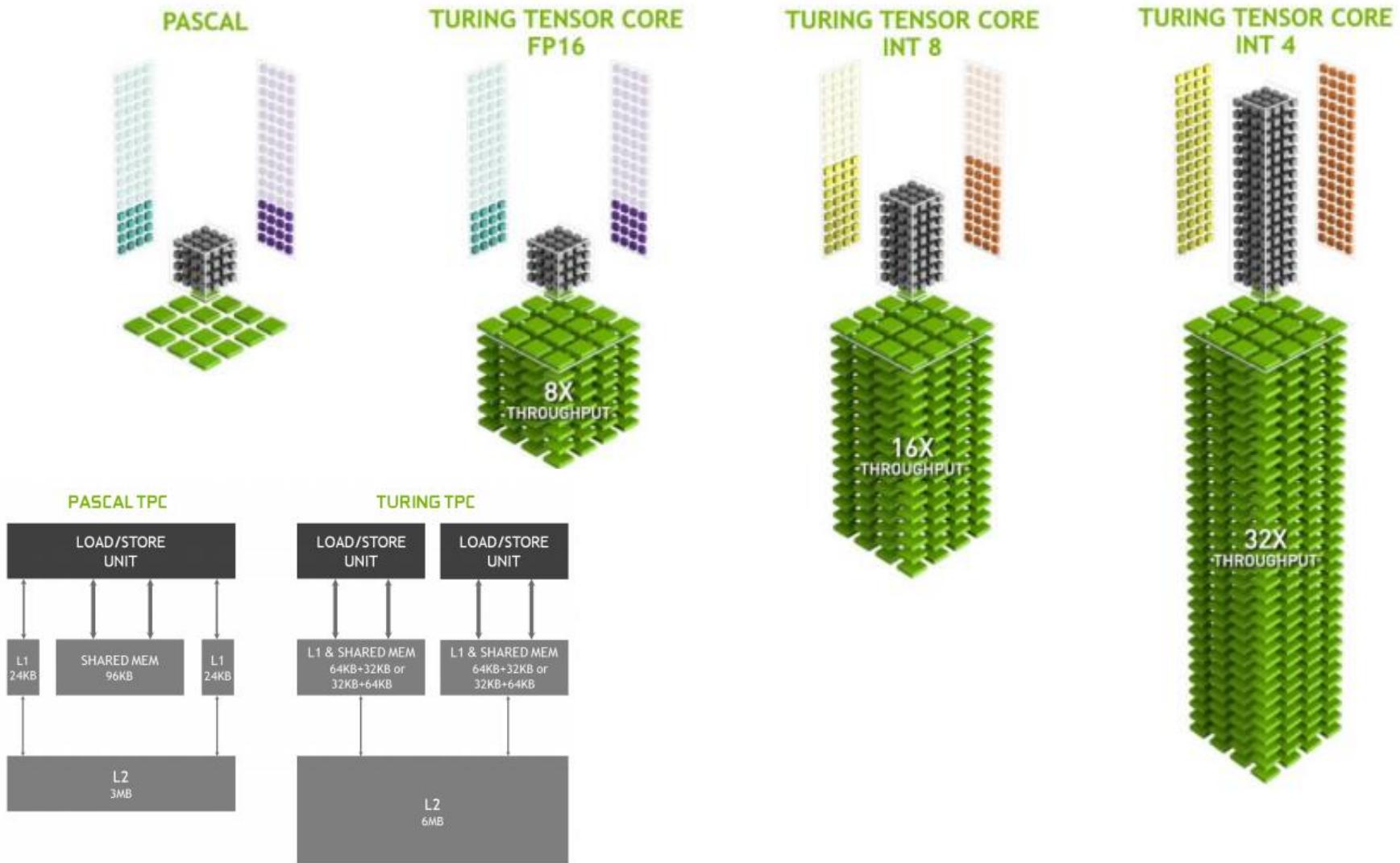
ADVANCED SHADING

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



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”

	DUAL XEON	DGX-1
FLOPS (CPU + GPU)	3 TF	170 TF
AGGREGATE NODE BW	76 GB/ s	768 GB/ s
ALEXNET TRAIN TIME	150 HOURS	2 HOURS
TRAIN IN 2 HOURS	>250 NODES*	1 NODE

*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
<https://software.intel.com/en-us/articles/caffe-training-on-multi-node-distributed-memory-systems-based-on-intel-xeon-processor-e5>

SYSTEM SPECIFICATIONS

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

NVIDIA DGX-2

DATA CENTER

PRODUCTS ▾

SOLUTIONS ▾

APPS ▾

FOR DEVELOPERS

TECHNOLOGIES ▾

DGX-2

OVERVIEW

TOUR

CONTACT US

NVIDIA DGX-2

The world's most powerful AI system for the most complex
AI challenges.



SYSTEM SPECIFICATIONS

GPUs	16X NVIDIA® Tesla V100
GPU Memory	512GB total
Performance	2 petaFLOPS
NVIDIA CUDA® Cores	81920
NVIDIA Tensor Cores	10240
NVSwitches	12
Maximum Power Usage	10kW
CPU	Dual Intel Xeon Platinum 8168, 2.7 GHz, 24-cores
System Memory	1.5TB
Network	8X 100Gb/sec Infiniband/100GigE Dual 10/25/40/50/100GbE
Storage	OS: 2X 960GB NVME SSDs Internal Storage: 30TB (8X 3.84TB) NVME SSDs
Software	Ubuntu Linux OS Red Hat Enterprise Linux OS See Software stack for details
System Weight	360 lbs (163.29 kgs)
Packaged System Weight	400lbs (181.44kgs)
System Dimensions	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
Operating Temperature Range	5°C to 35°C (41°F to 95°F)

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 Mac OS

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

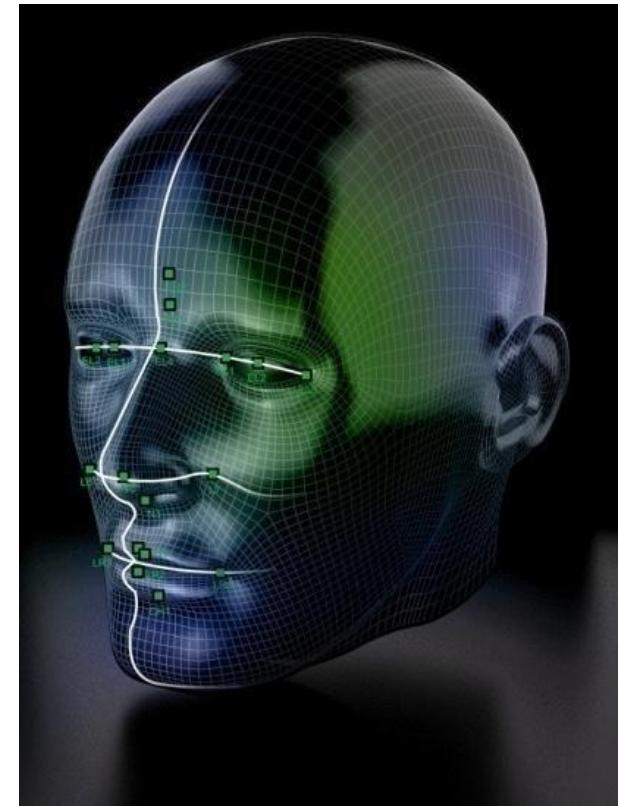
- Each host thread can now access all GPUs in the system
 - One thread per GPU limitation removed
- Easier than ever for applications to take advantage of multi-GPU
 - Single-threaded applications can now benefit from multiple GPUs
 - Easily coordinate work across multiple GPUs

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 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

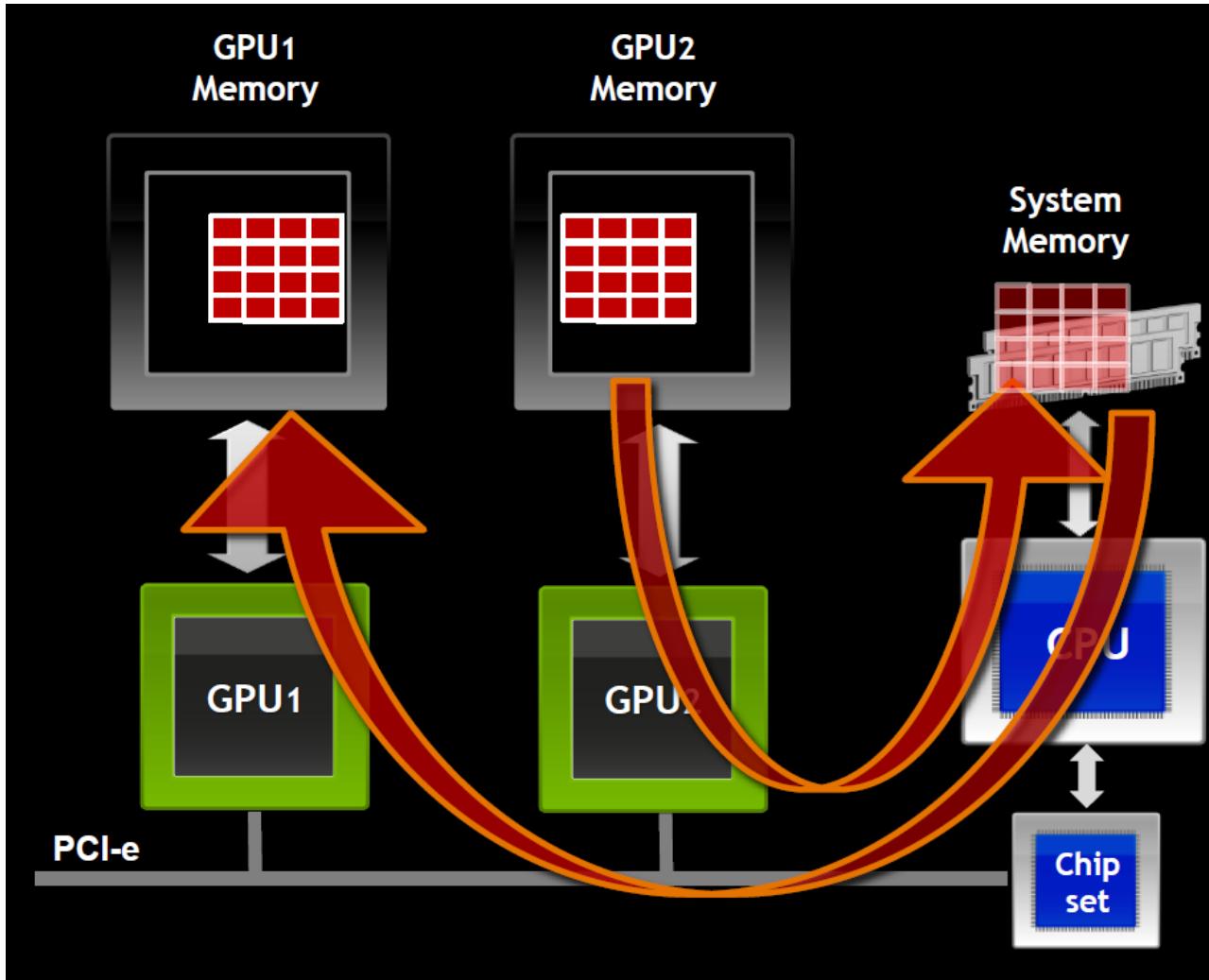
Version 2.0

for applications that communicate within a node

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

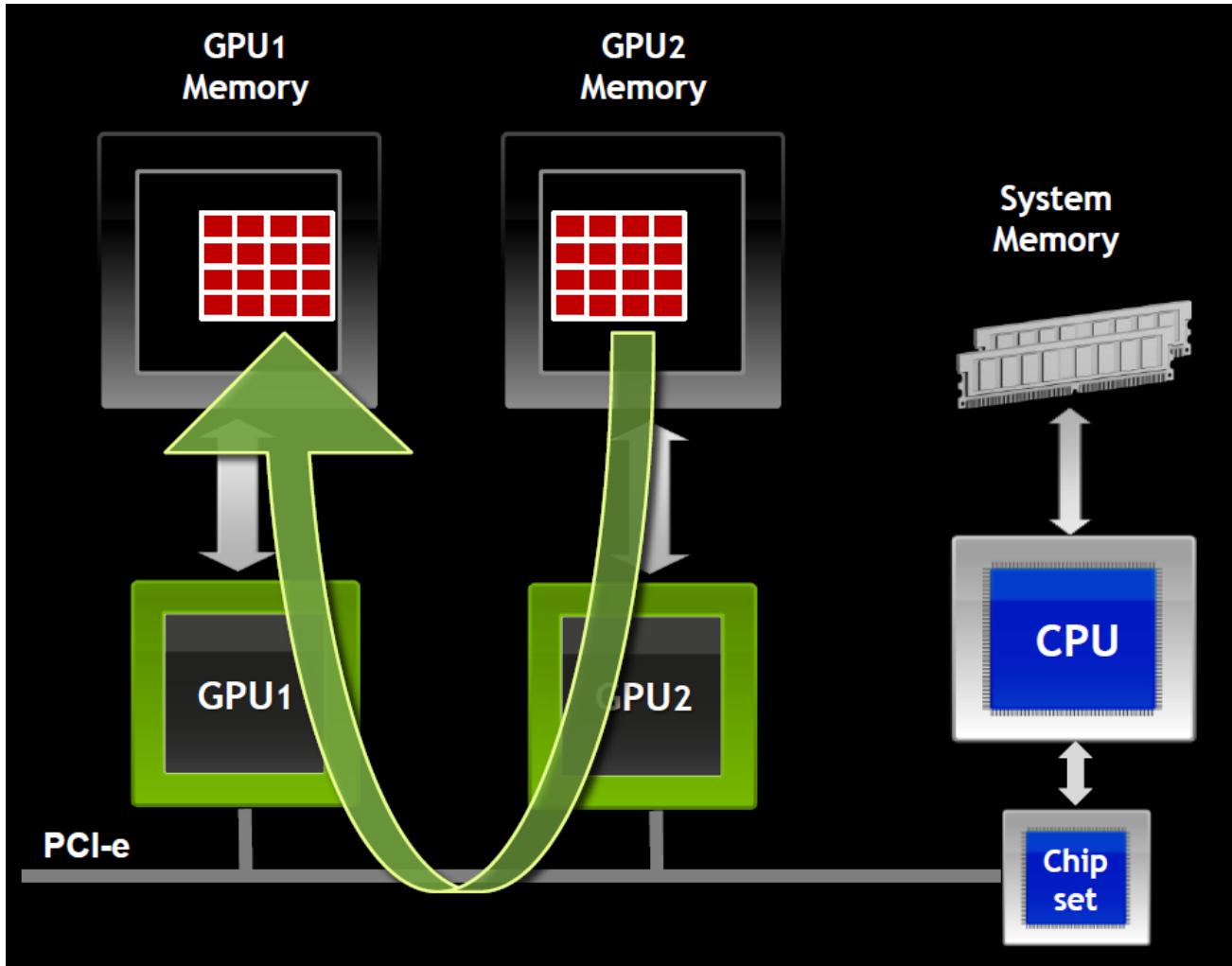
Before GPUDirect 2.0

Two copies required



GPUDirect 2.0: Peer-to-Peer Communication

Only one copy required

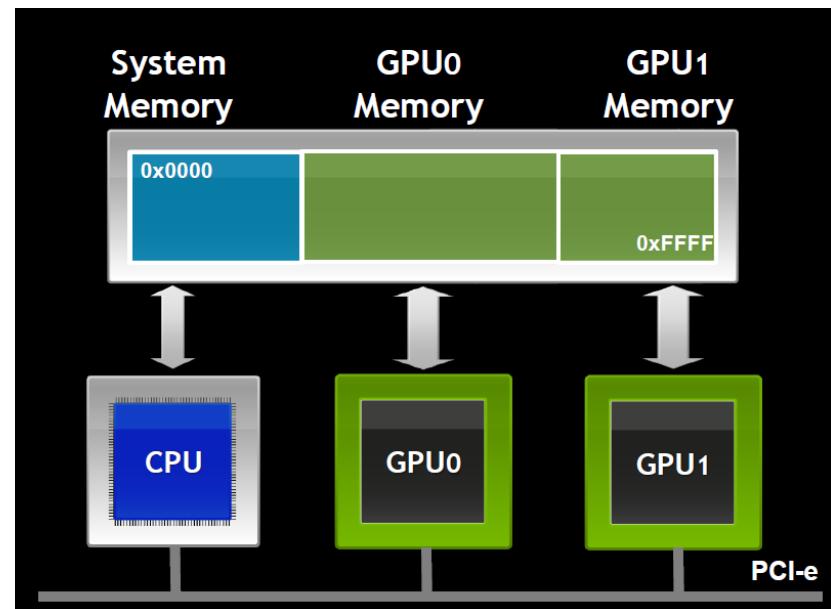
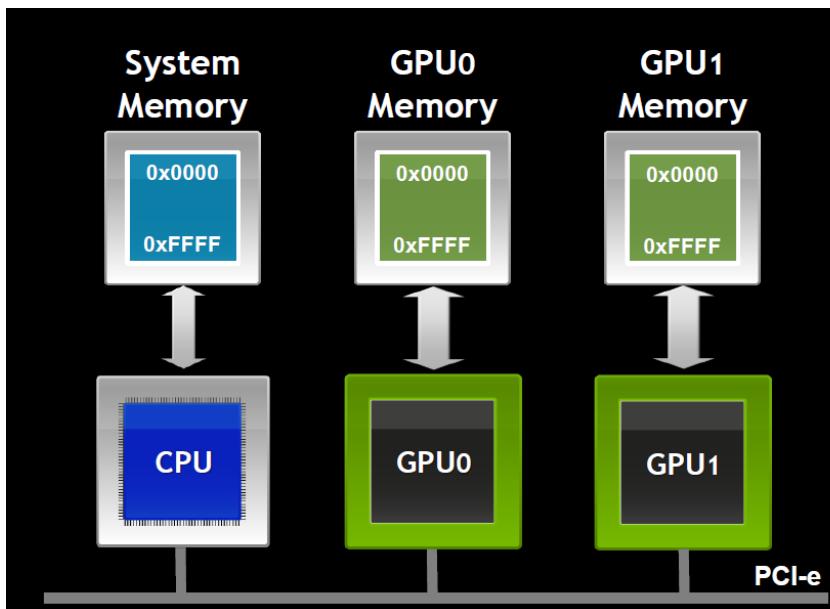


GPUDirect 2.0: Peer-to-Peer Communication

- 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

Before UVA	With UVA
Separate options for each permutation	One function handles all cases
<code>cudaMemcpyHostToHost</code> <code>cudaMemcpyHostToDevice</code> <code>cudaMemcpyDeviceToHost</code> <code>cudaMemcpyDeviceToDevice</code>	<code>cudaMemcpyDefault</code> (data location becomes an implementation detail)

New Developer Tools

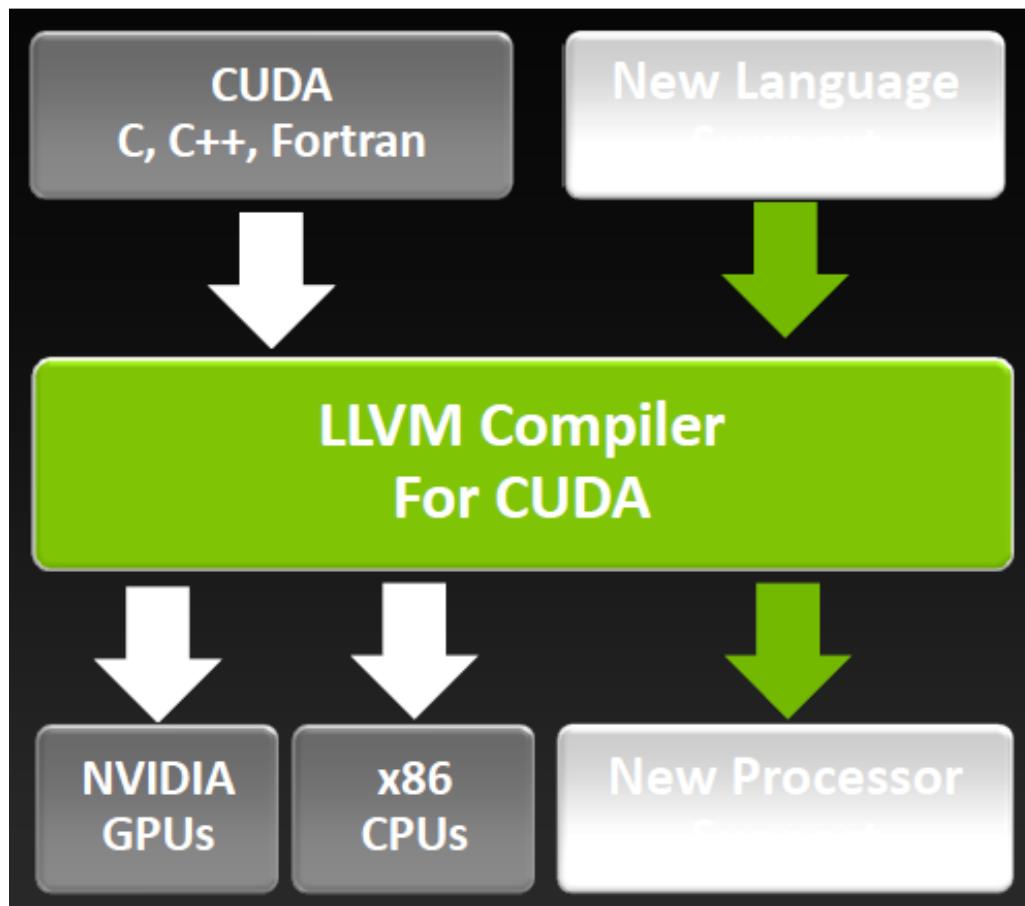
- Auto Performance Analysis: Visual Profiler
 - Identify limiting factor
 - Analyze instruction throughput
 - Analyze memory throughput
 - Analyze kernel occupancy
- C++ Debugging
 - cuda-gdb for MacOS
- GPU Binary Disassembler

CUDA 5.0

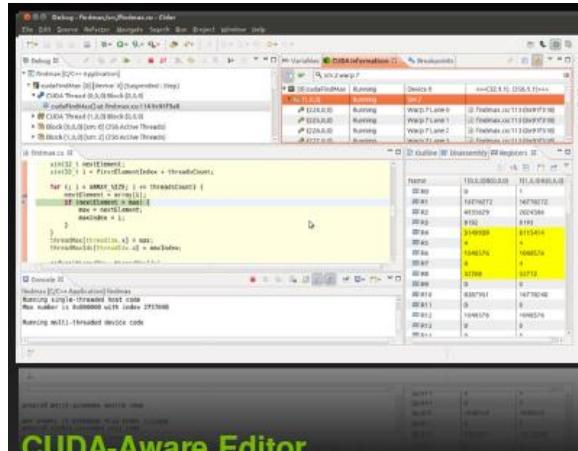
Mark Harris
Chief Technologist, GPU
Computing

Open Source LLVM Compiler

- Provides ability for anyone to add CUDA to new languages and processors

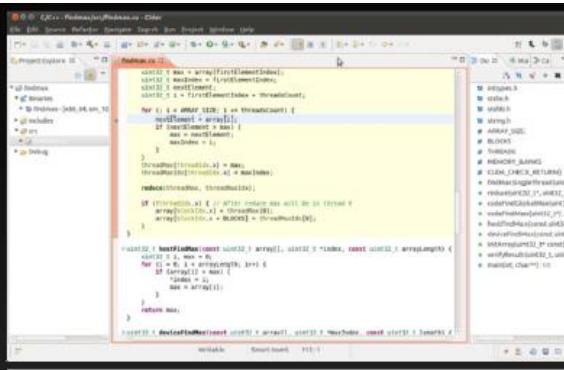


NVIDIA Nsight, Eclipse Edition



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

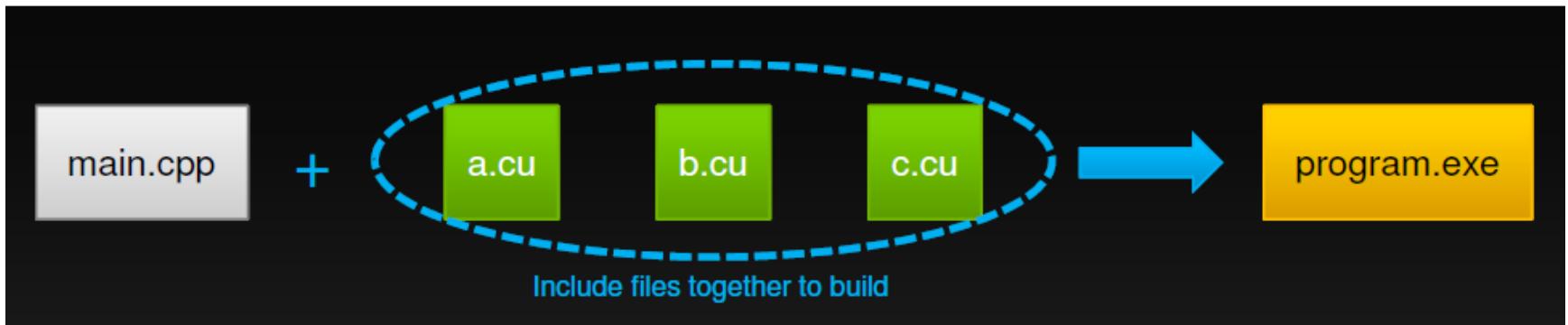


Nsight Profiler

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

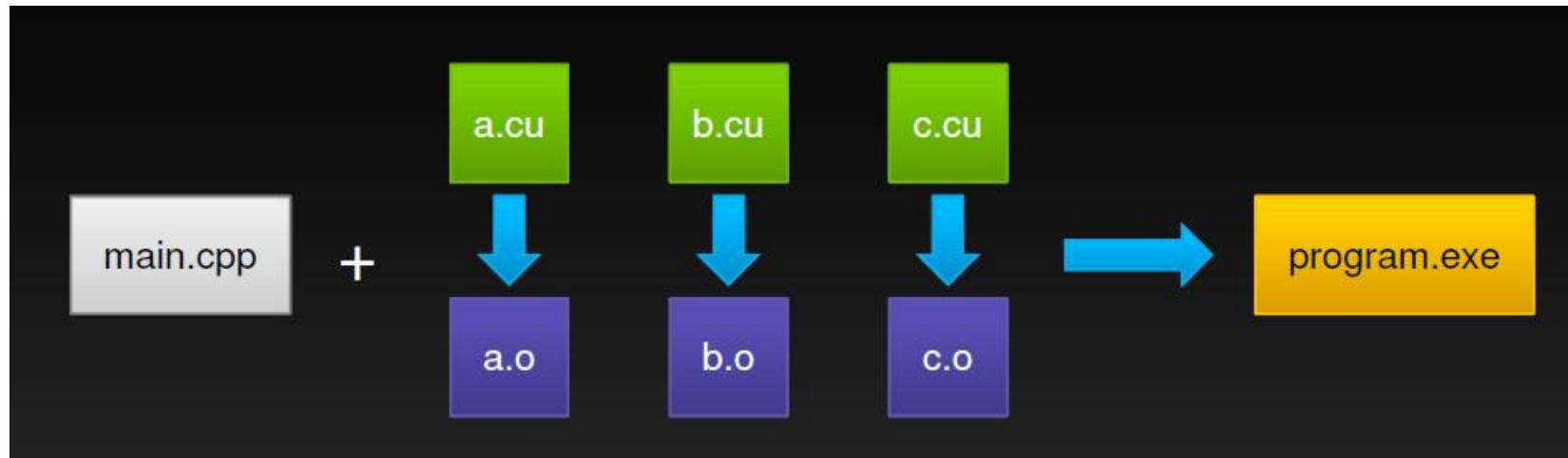
For Linux and Mac OS

CUDA 4: Whole-Program Compilation & Linking



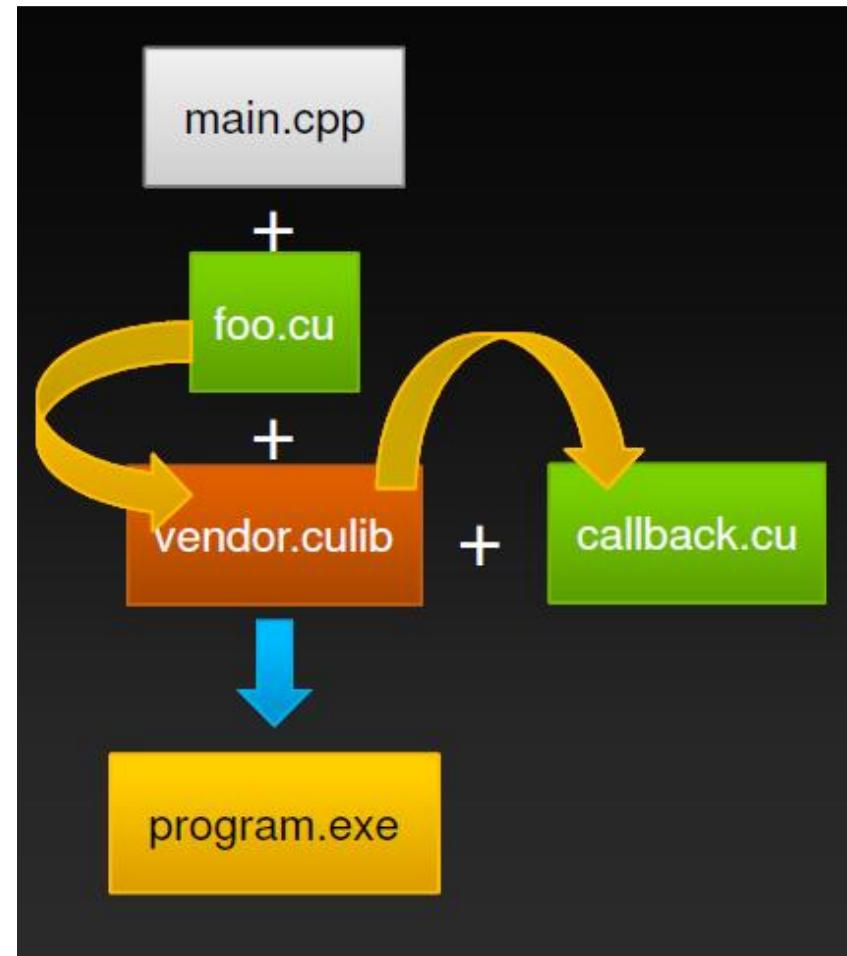
CUDA 5: GPU Library Object Linking

- 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



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

```
__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)

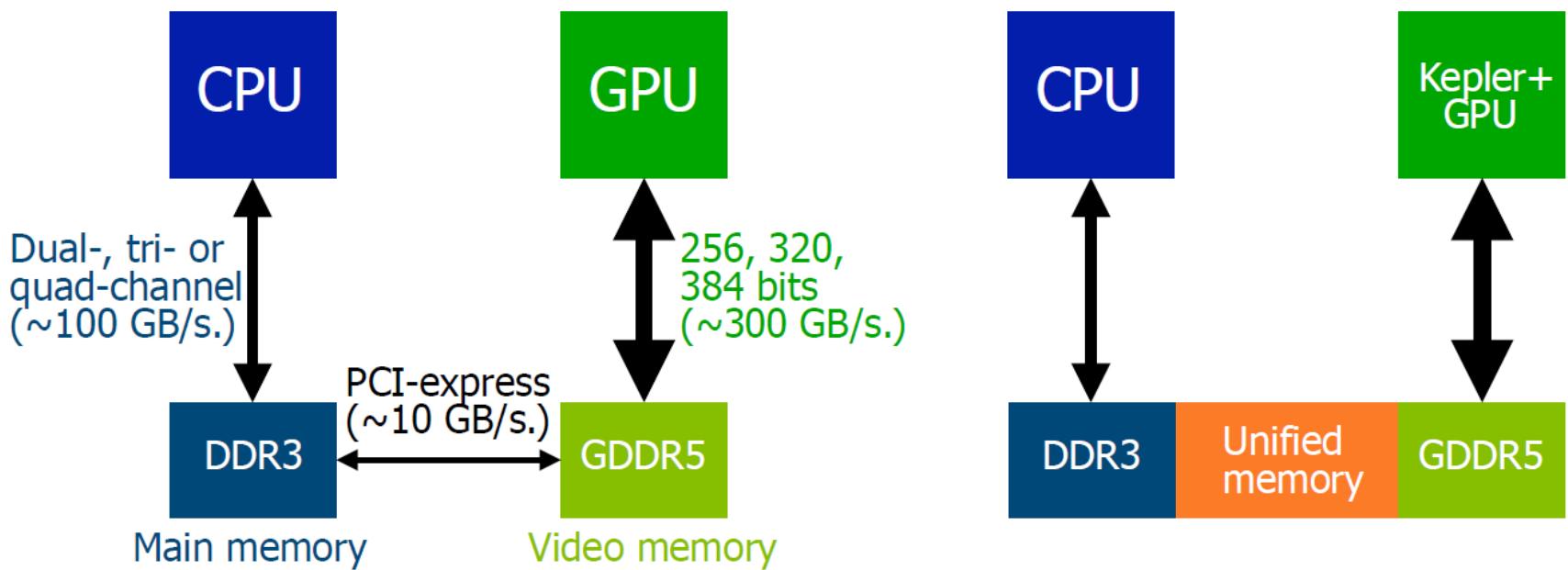
CUDA 6 Highlights

- 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



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

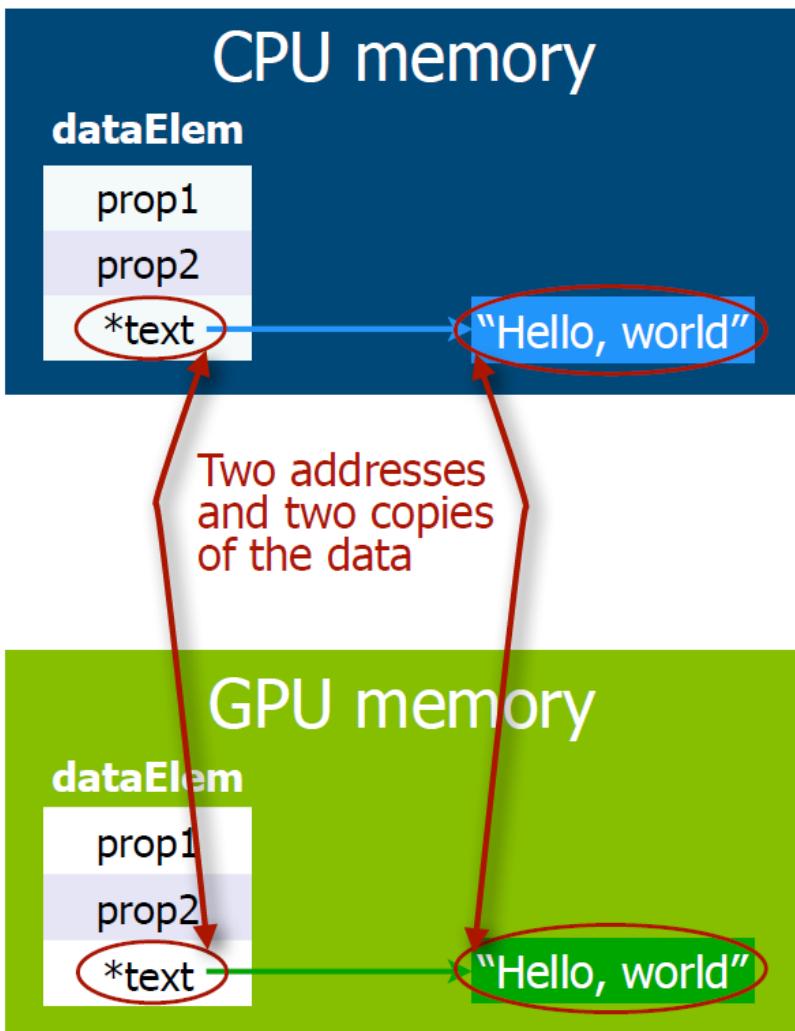
Memory Types

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

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



```
void launch(dataElem *elem) {  
    dataElem *g_elem;  
    char *g_text;  
  
    int textlen = strlen(elem->text);  
  
    // Allocate storage for struct and text  
    cudaMalloc(&g_elem, sizeof(dataElem));  
    cudaMalloc(&g_text, textlen);  
  
    // Copy up each piece separately, including  
    // new "text" pointer value  
    cudaMemcpy(g_elem, elem, sizeof(dataElem));  
    cudaMemcpy(g_text, elem->text, textlen);  
    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

CPU memory

Unified memory

dataElem

prop1

prop2

*text

→ "Hello, world"

GPU memory

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

- 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

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;

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

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

    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

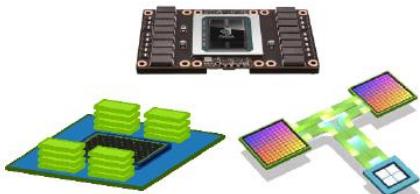
CUDA 8.0

By Milind Kukanur
NVIDIA

What's New

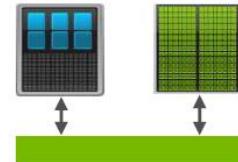
PASCAL SUPPORT

New Architecture
NVLINK
HBM2 Stacked Memory
Page Migration Engine



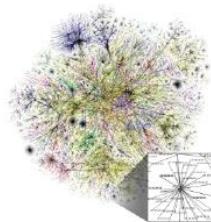
UNIFIED MEMORY

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

```
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

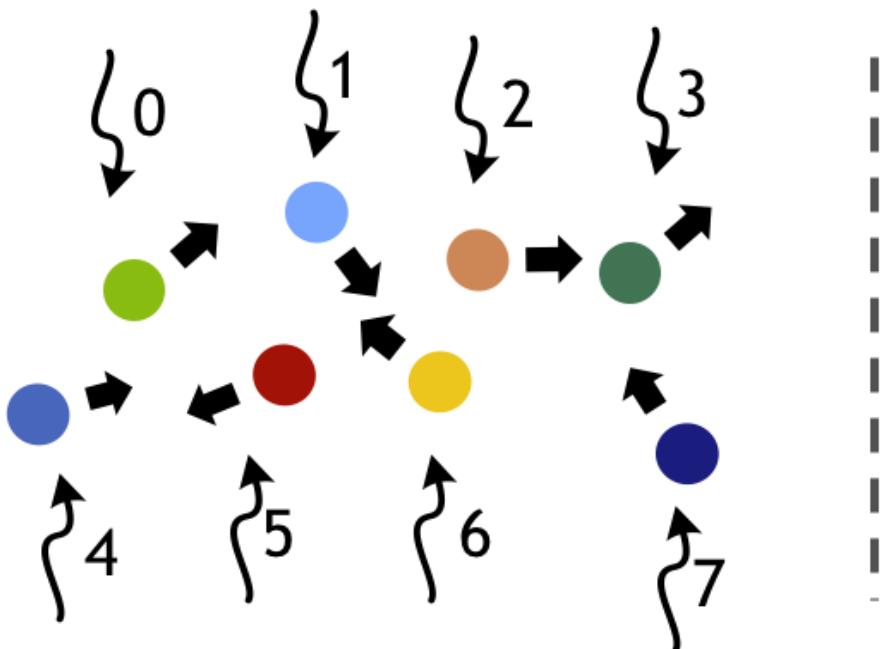
- Support for Volta
- Cooperative groups
- Tensor Core API
- New Visual Profiler
- Support for C++ 14

Cooperative Groups

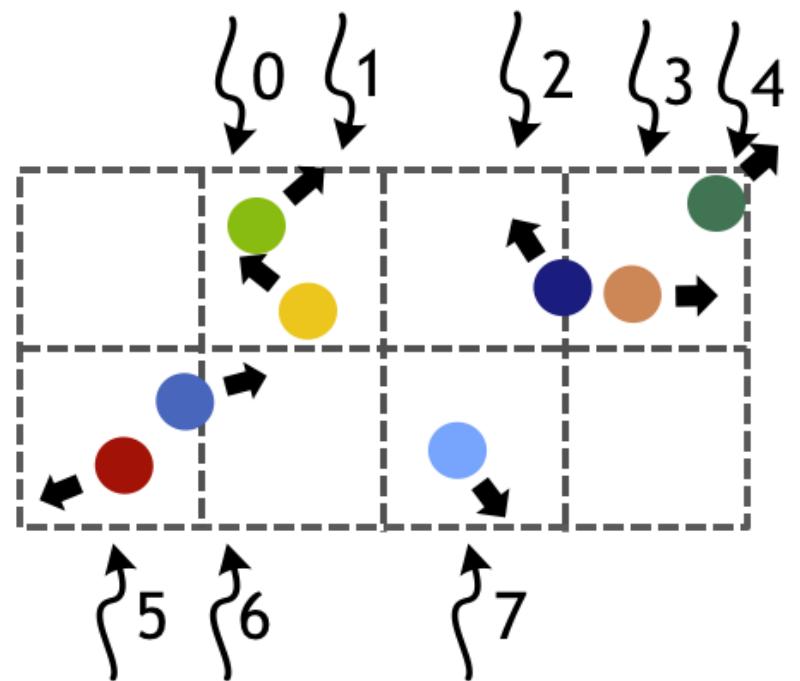
- Ability to define groups of threads explicitly at sub-block and multiblock granularities

```
__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);
```

New Implementation

```
__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

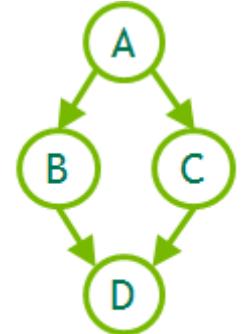
New Features

- Support for Turing
- CUDA graphs
- New asynchronous task-graph programming model
- New profiler and debugger

New Turing Warp Matrix Functions

	Input Precision	Output	Supported Sizes	Max Ops/Clock/SM	
Native Types	half	half or float	16 x 16 x 16 32 x 8 x 16 8 x 32 x 16	1024	
	char	integer (int32)		2048	
	unsigned char				
Experimental	precision::u4 (4-bit unsigned)	integer (int32)	8 x 8 x 32 8 x 8 x 128	4096	
	precision::s4 (4-bit signed)			16384	
	precision::b1 (1-bit)				

CUDA graphs



```
// Define graph of work + dependencies
```

```
cudaGraphCreate(&graph);  
                    Workflow 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 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

```
#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

```
#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

```
#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);
```

for

```
#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

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

- Or, equivalently

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

Specifying Number of Threads

```
#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

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

- 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

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

Scheduling

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

```
#pragma omp for schedule(dynamic, 3)
for(int n=0; n<10; ++n) printf(" %d", n);
printf(".\n");
```

ordered

```
#pragma omp for ordered schedule(dynamic)
for(int n=0; n<100; ++n)
{
    files[n].compress();
}

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

reduction

```
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.

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

simd

```
#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

```
#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

```
#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:

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

```
#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

- 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

```
#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

```
#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**

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