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

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

Outline

- Introduction to OpenMP

- Hardware developments
- Developments in CUDA

OpenMP

Based on tutorial by Joel Yliluoma

http://bisqwit.iki.fi/story/howto/openmp/

OpenMP in C++

- OpenMP consists of a set of compiler #pragmas that control how the program works.
- The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.

Simple Example

- Multiple threads

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

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

  // the table is now initialized
}
```

Simple Example

- Single thread, SIMD

```
#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

```
#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 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);
  }
```

The Fermi Architecture

Selected notes from presentation by:

Michael C. Shebanow

Principal Research Scientist,
NV Research
mshebanow@nvidia.com

(2010)

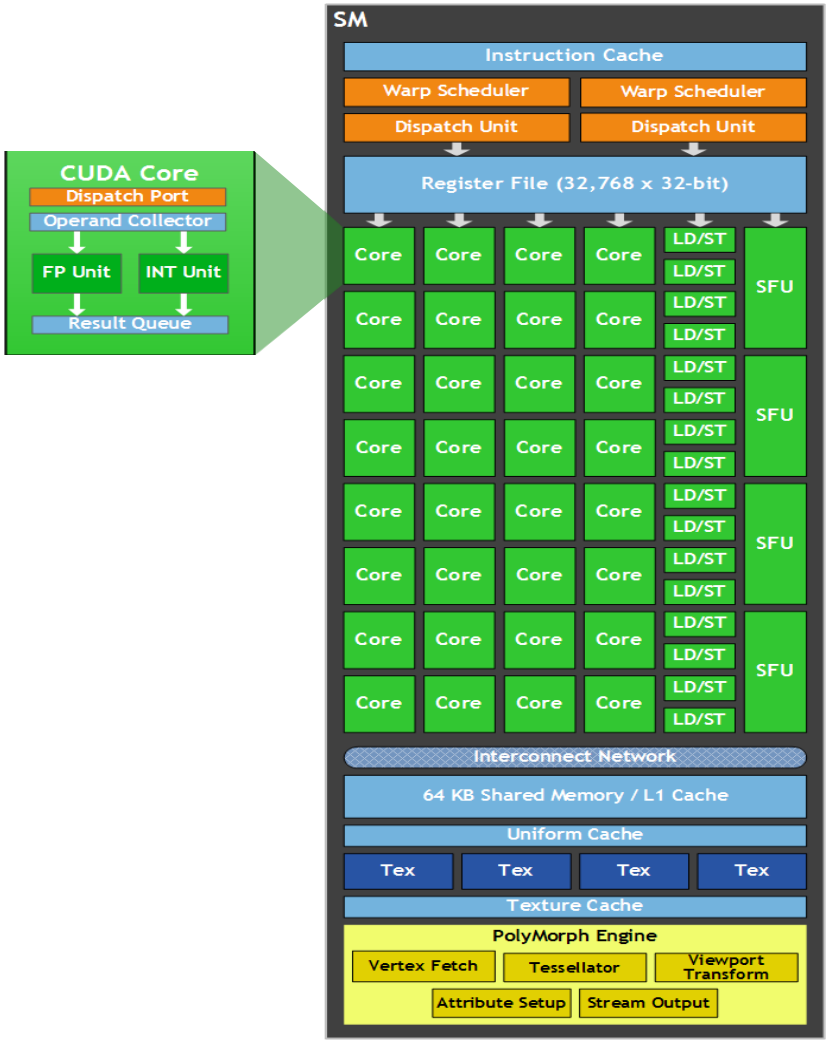
Much Better Compute

- 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

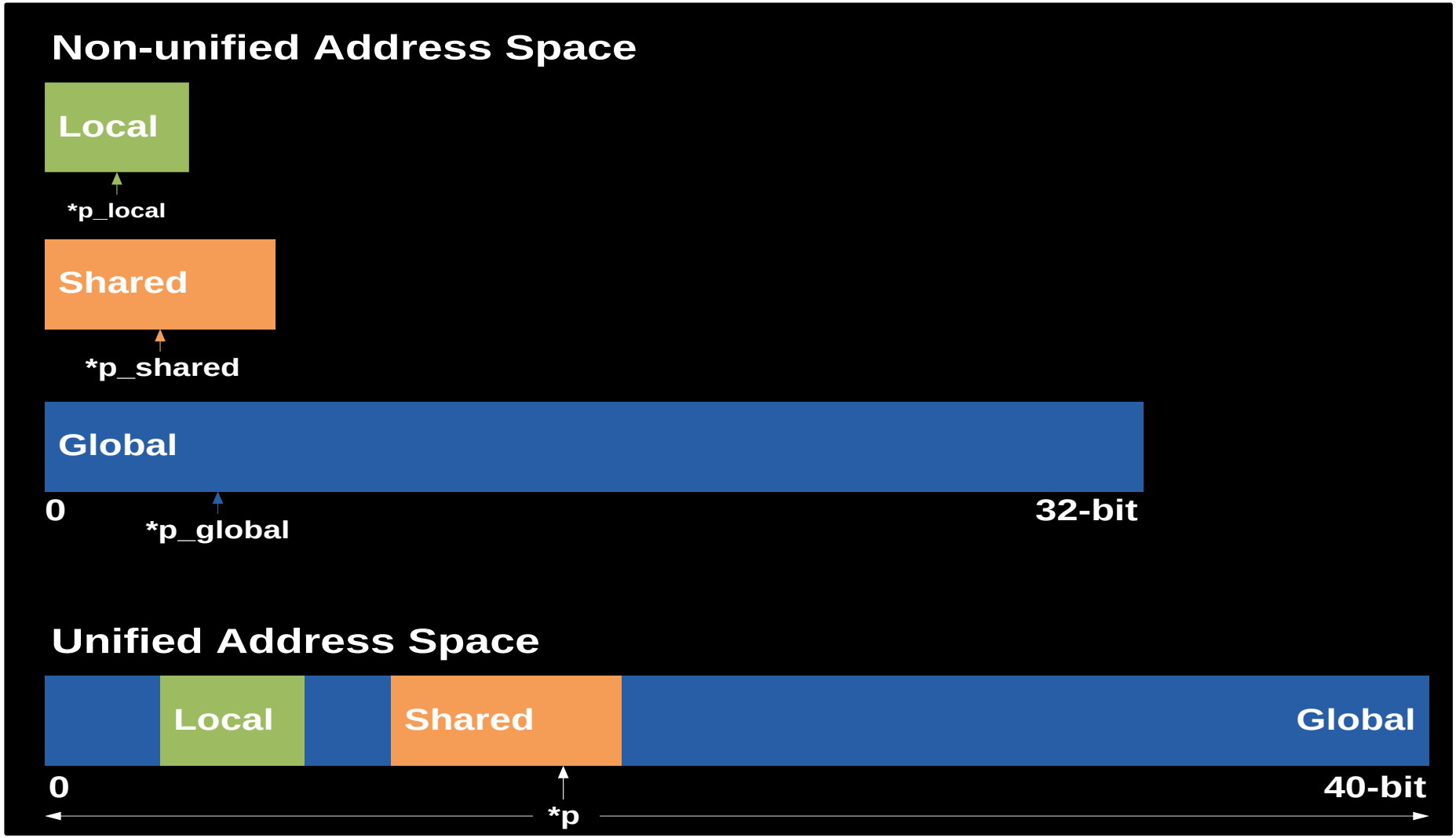
	GT200	GF100	Benefit
L1 Texture Cache (per quad)	12 KB	12 KB	Fast texture filtering
Dedicated L1 LD/ST Cache	X	16 or 48 KB	Efficient physics and ray tracing
Total Shared Memory	16KB	16 or 48 KB	More data reuse among threads
L2 Cache	256KB (TEX read only)	768 KB (all clients read/write)	Greater texture 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.

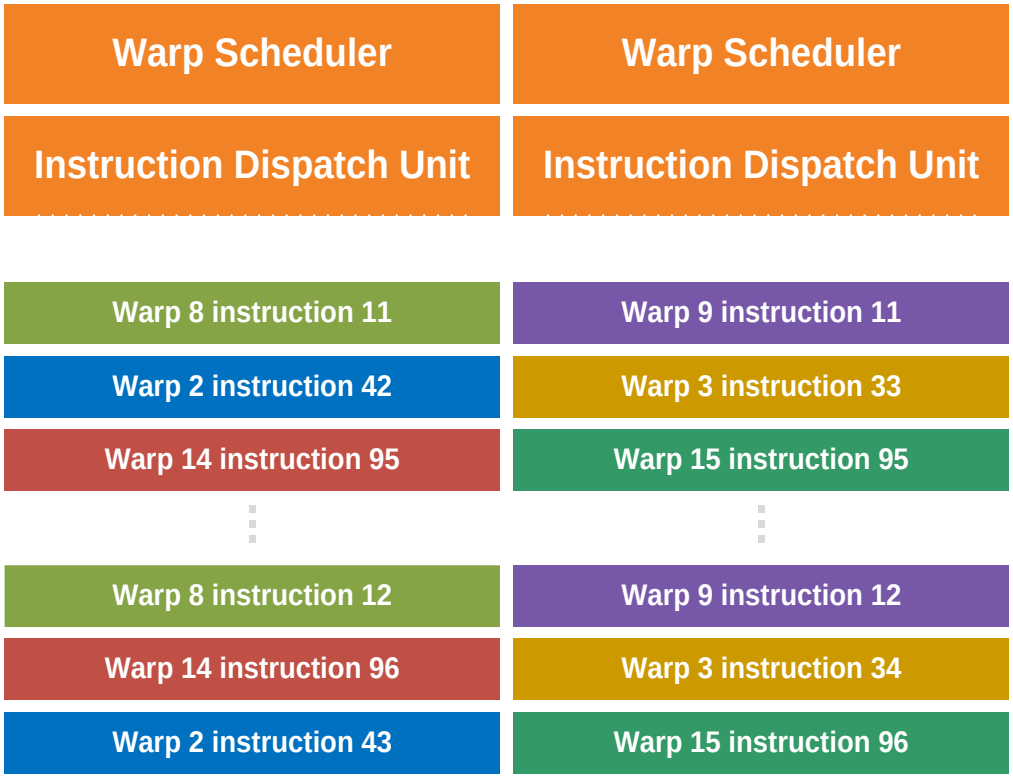


Unified Load/Store Addressing



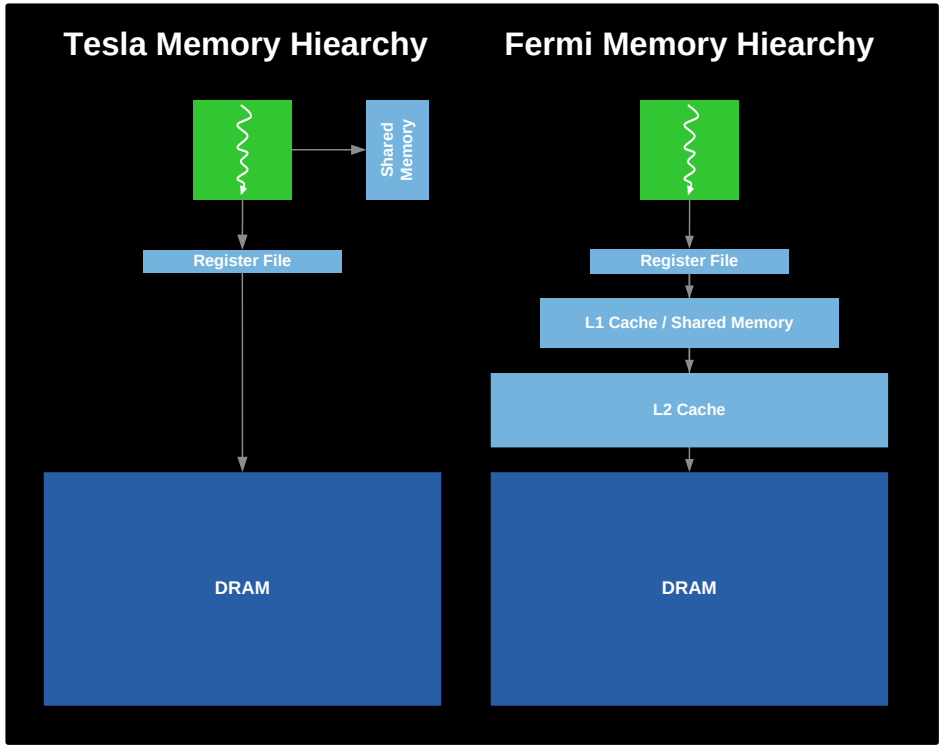
Instruction Issue and Control Flow

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



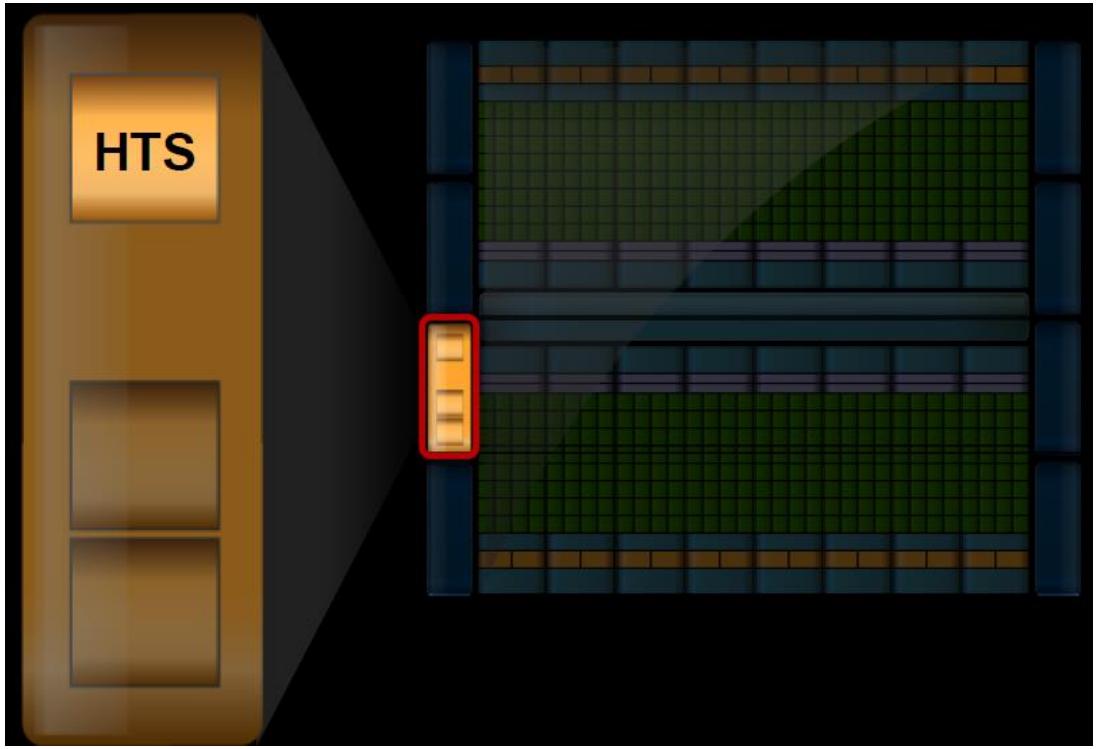
Caches

- 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



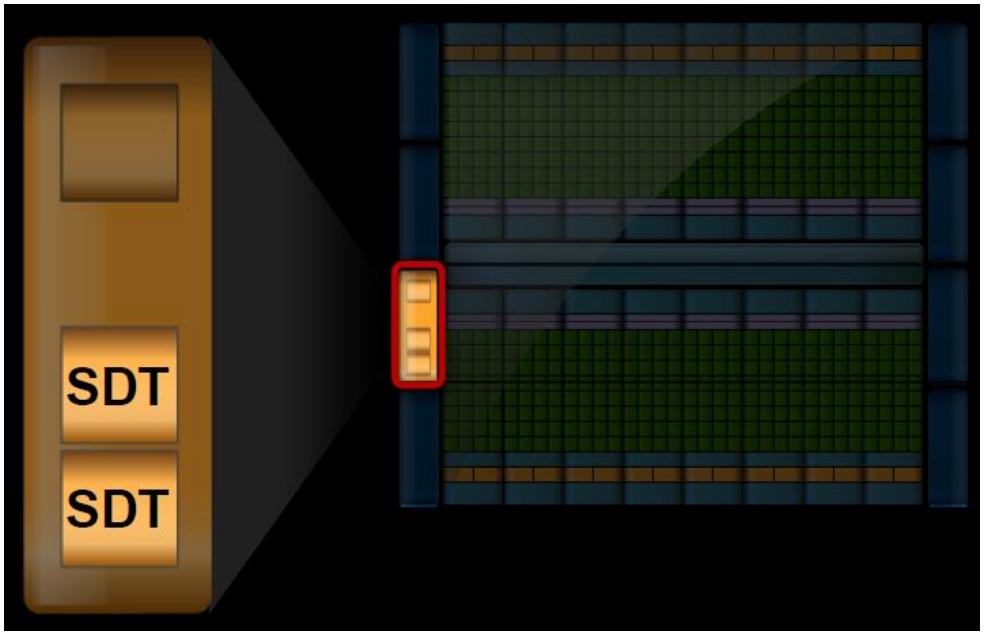
GigaThread Hardware Thread Scheduler

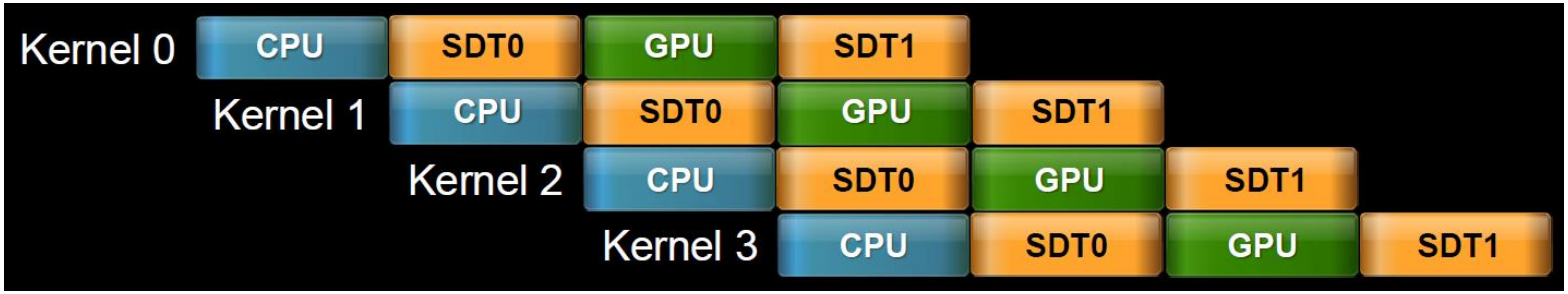
- Hierarchically manages tens of thousands of simultaneously active threads
- 10x faster context switching on Fermi
- Concurrent kernel execution



GigaThread Streaming Data Transfer Engine

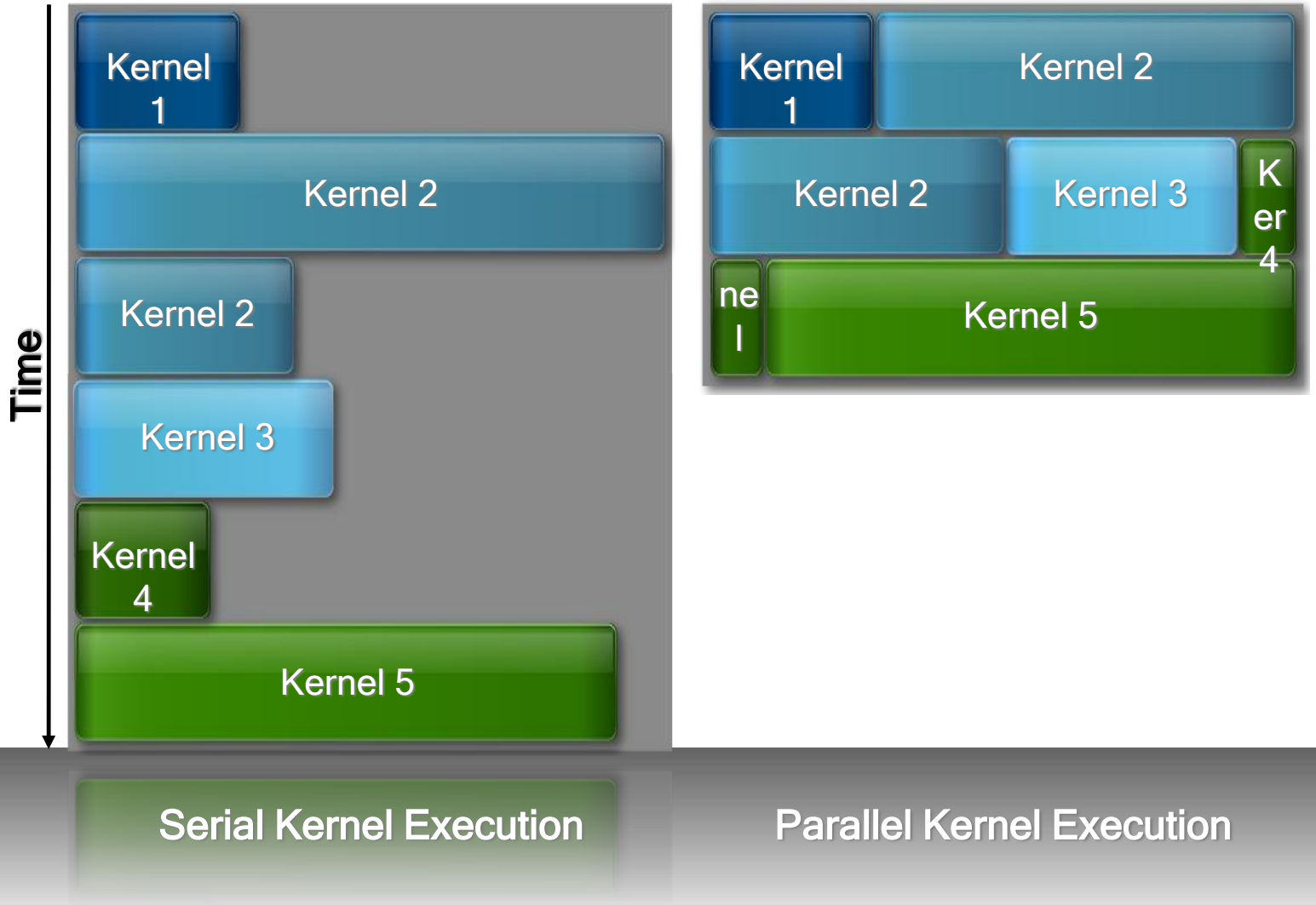
- Dual DMA engines
- Simultaneous CPU→GPU and GPU→CPU data transfer
- Fully overlapped with CPU/GPU processing





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

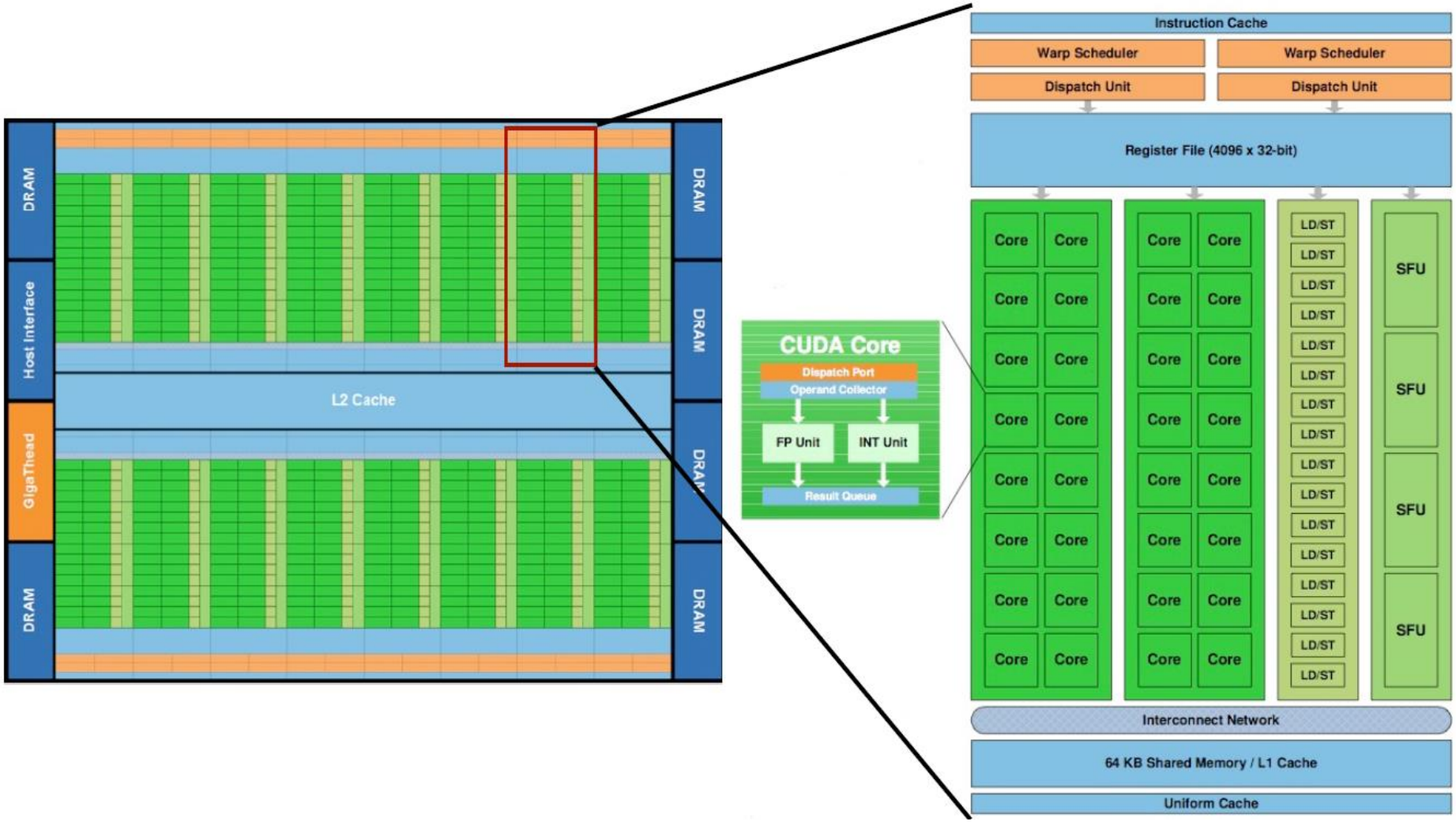
Hardware Resources and Peak Performance

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

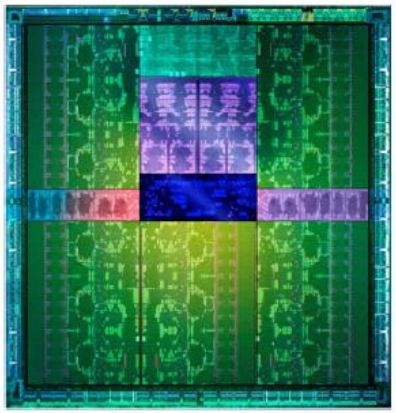
Memory Features

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

Fermi

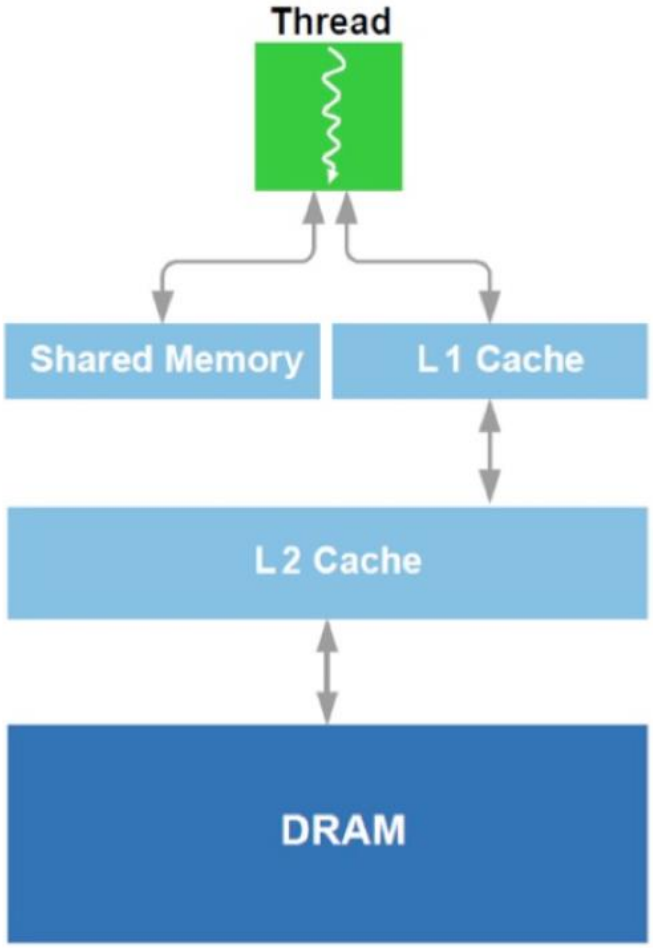

DRAM
Host Interface
GigaThead
DRAM
L2 Cache
DRAM
DRAM
DRAM
DRAM
CUDA Core
Dispatch Port
Operand Collector
FP Unit
INT Unit
Result Queue
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (4096 x 32-bit)
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache

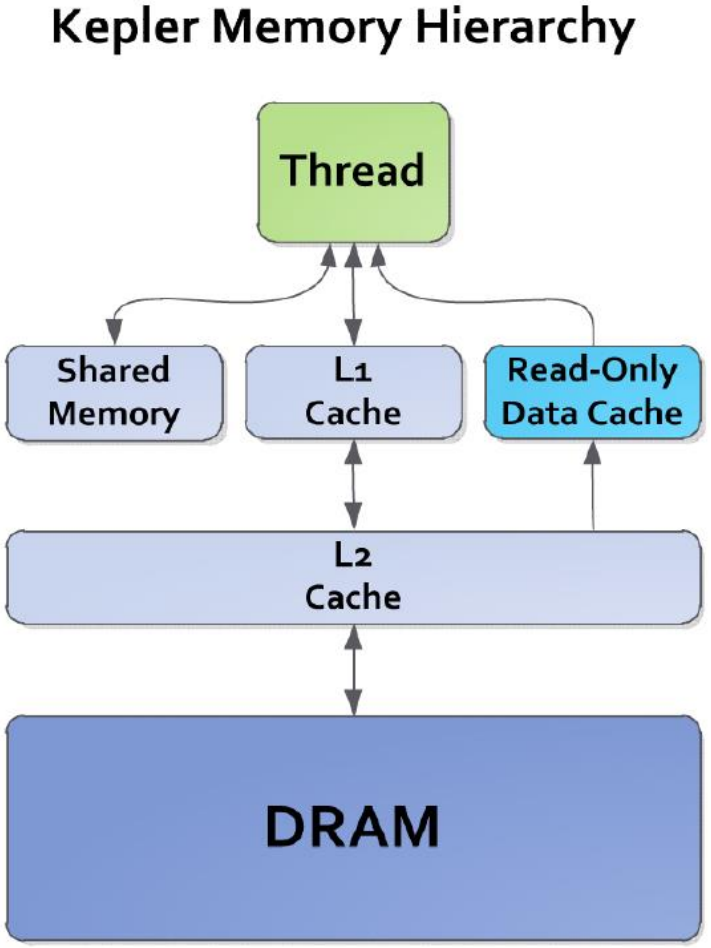
Kepler GK110



From SM to SMX in Kepler

Differences in 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.

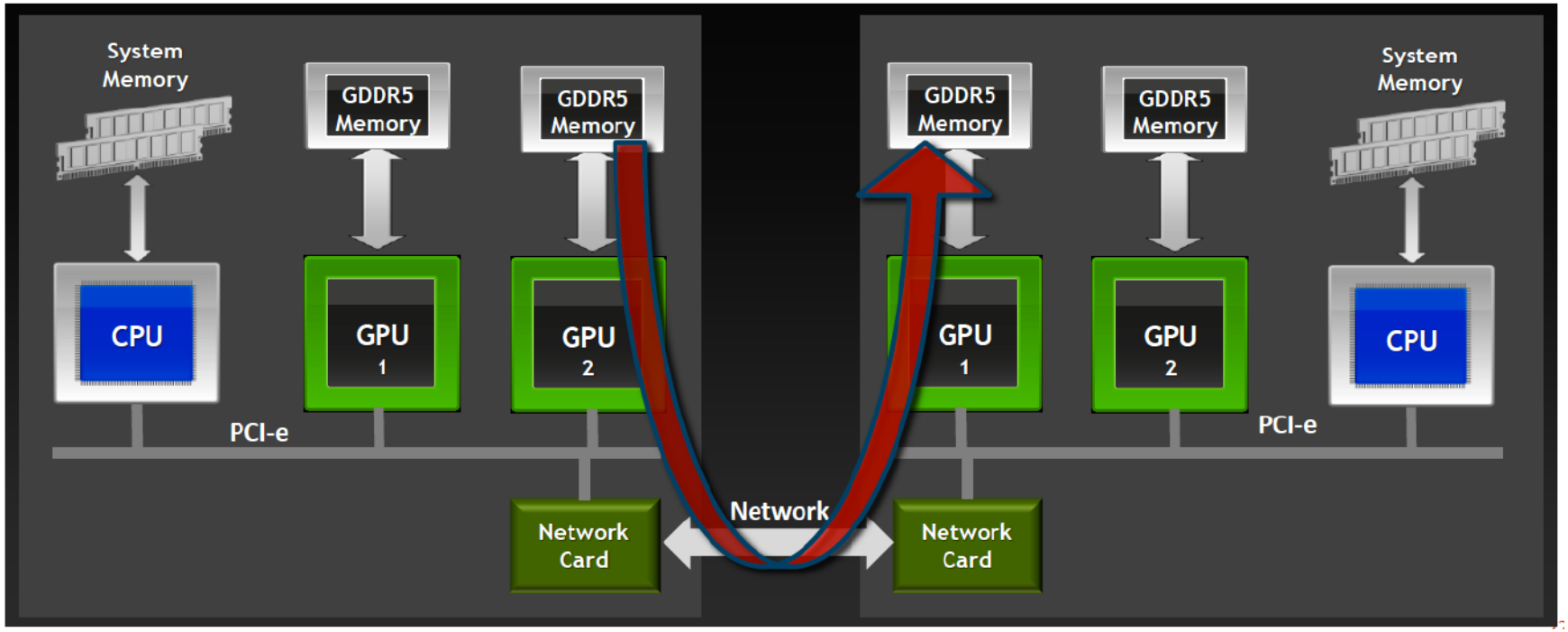
How to use Data Cache

- 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

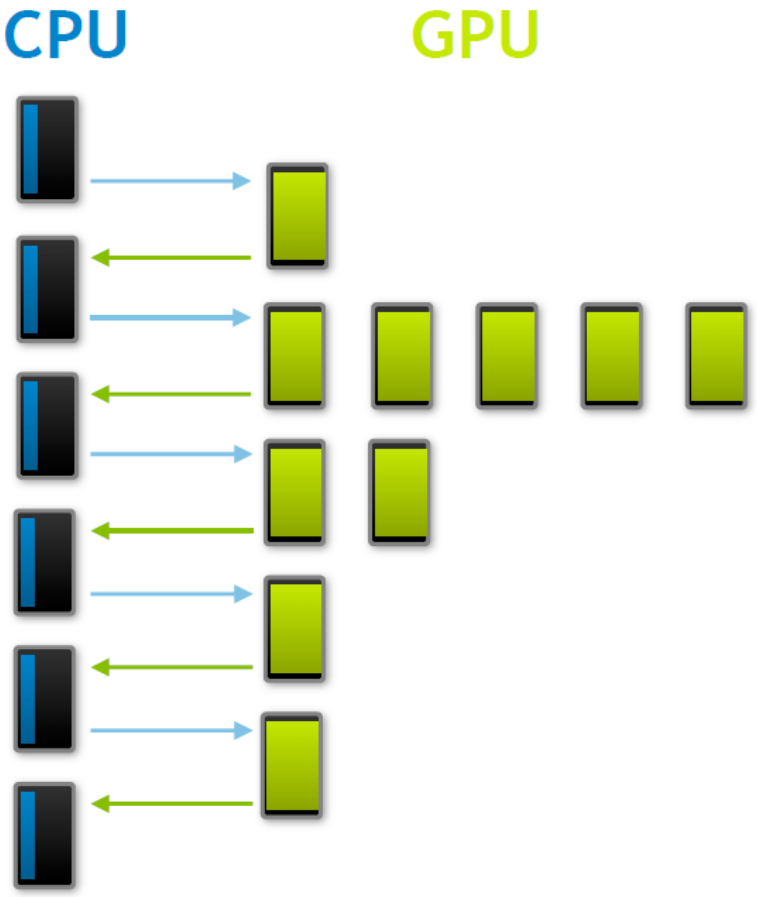
- The ability to launch new grids from the GPU:
  - Dynamically: Based on run-time data
  - Simultaneously: From multiple threads at once
  - Independently: Each thread can launch a different grid

Fermi: Only CPU
can generate GPU work.

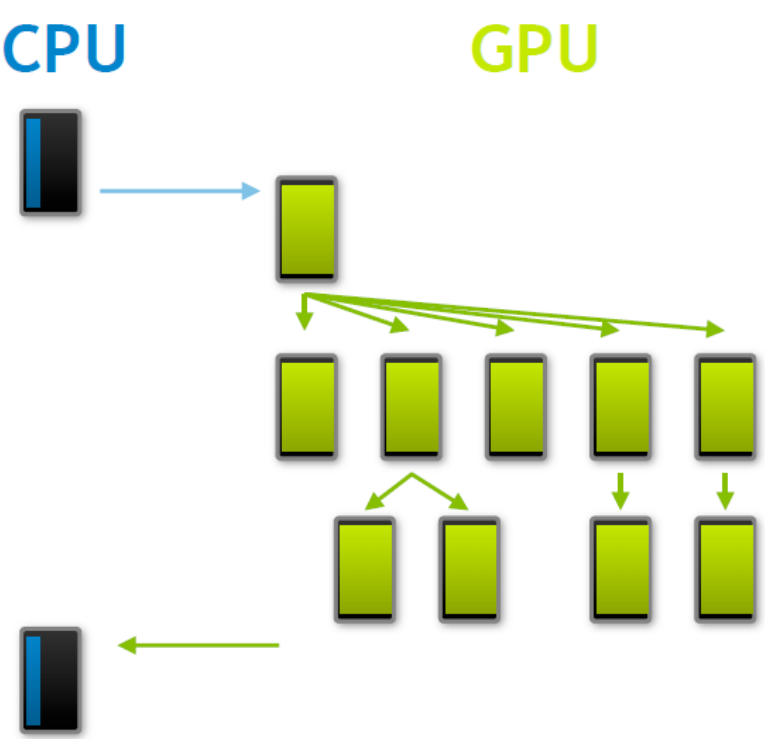
Kepler: GPU can
generate work for itself.

Dynamic Parallelism

The pre-Kepler GPU is a co-processor



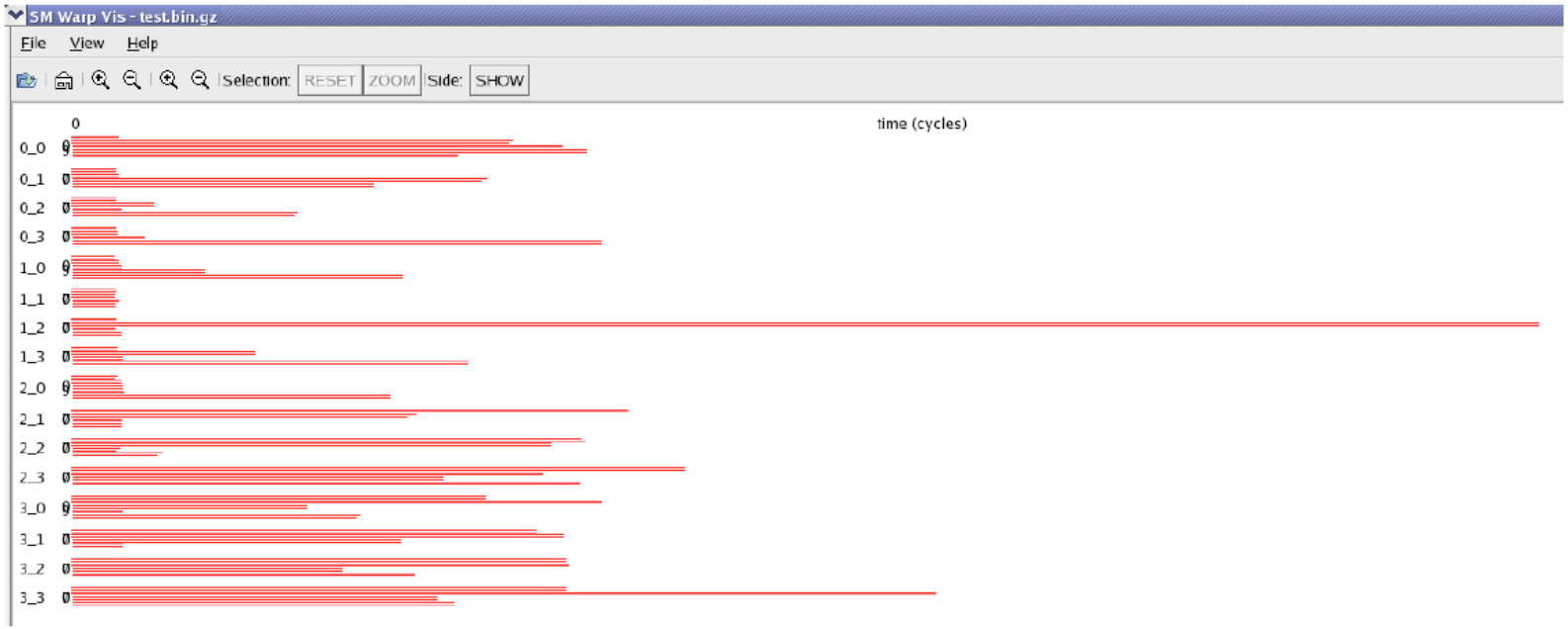
The Kepler GPU is autonomous:
Dynamic parallelism



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

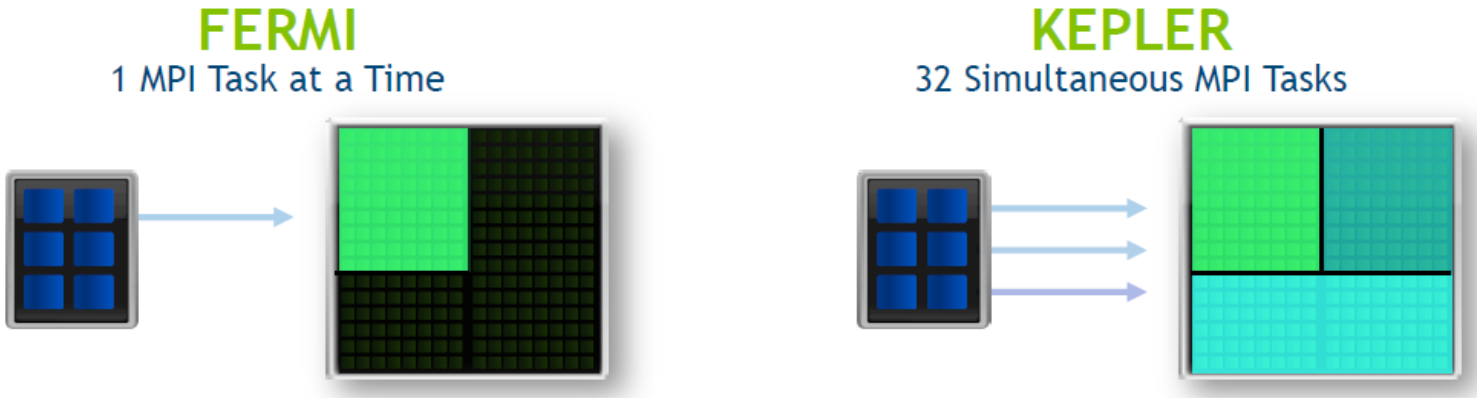
Workload Balance

- 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:

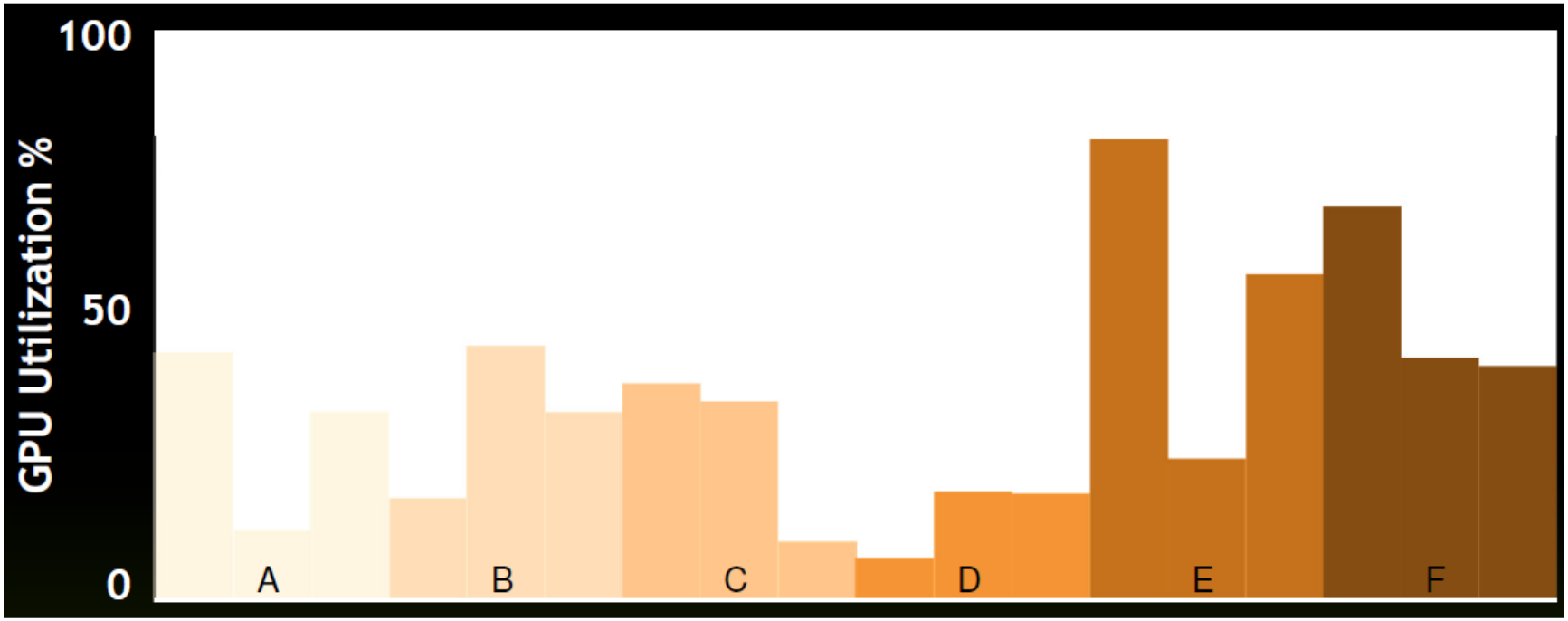


Hyper-Q

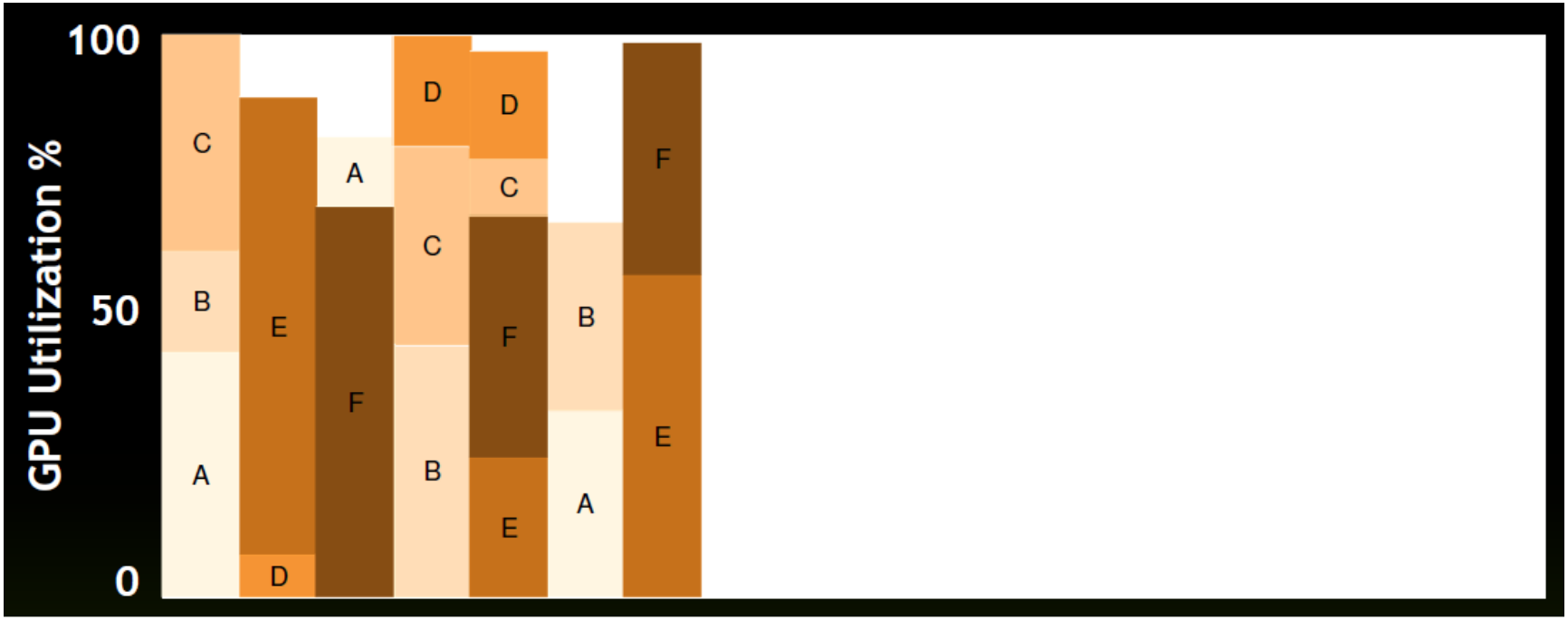
- In Fermi, several CPU processes can send thread blocks to the same GPU, but a kernel cannot start its execution until the previous one has finished
- In Kepler, we can execute simultaneously up to 32 kernels launched from different:
  - MPI processes, CPU threads (POSIX threads) or CUDA streams
- This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



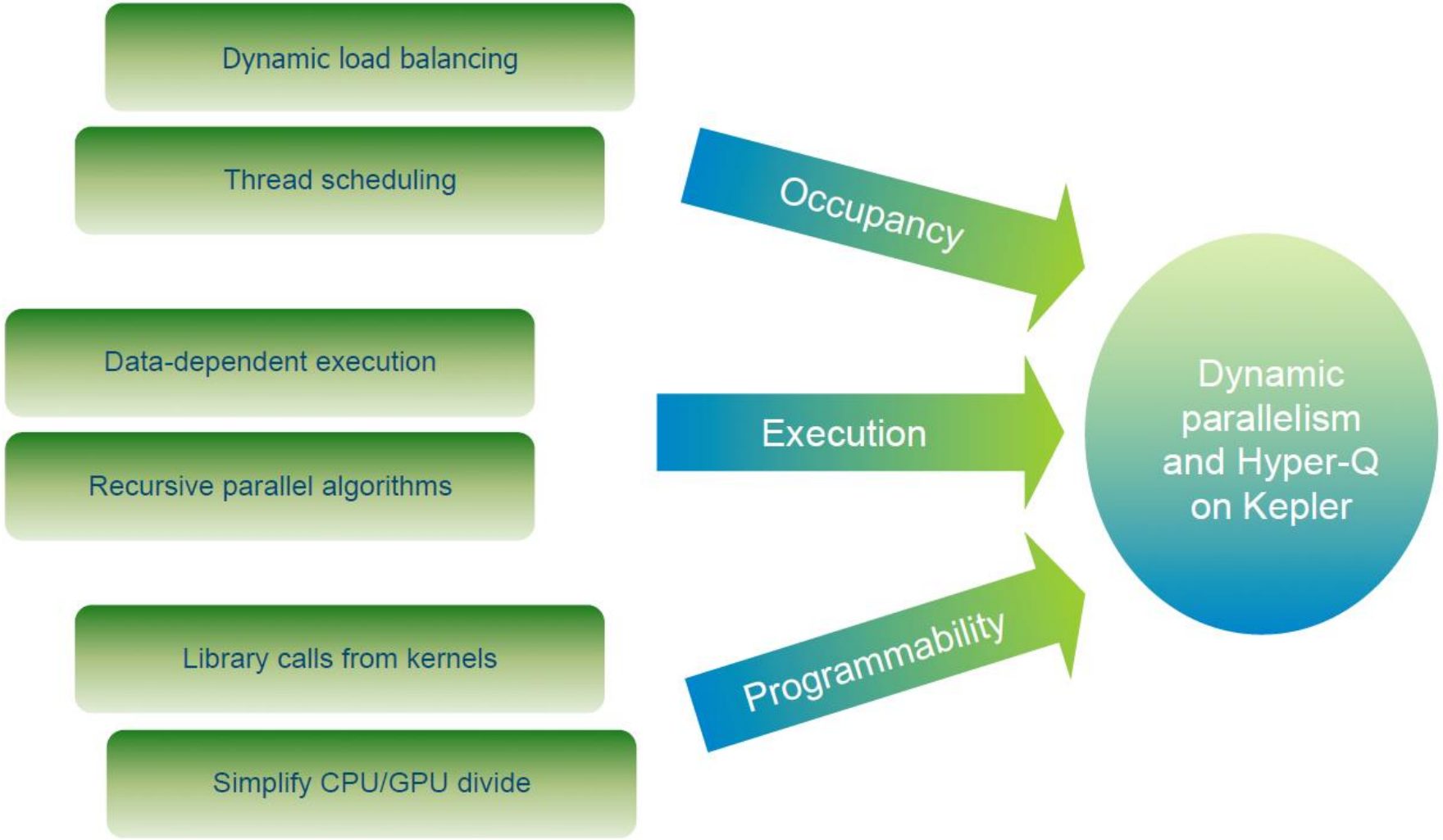
Without Hyper-Q



With Hyper-Q

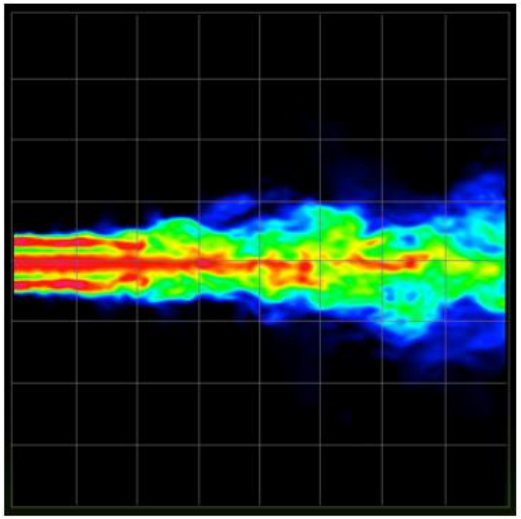


Six Ways to Improve Code on Kepler



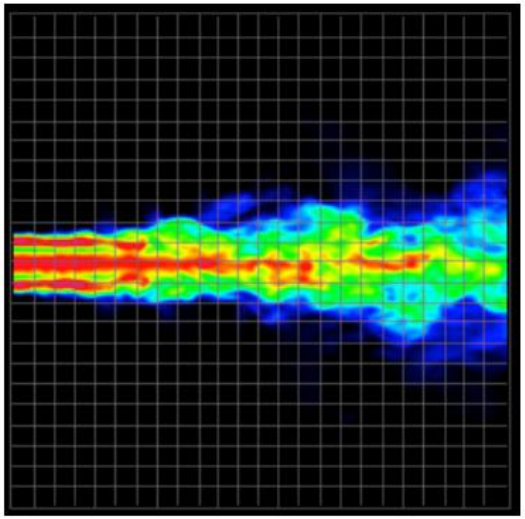
Dynamic Work Generation

Coarse grid



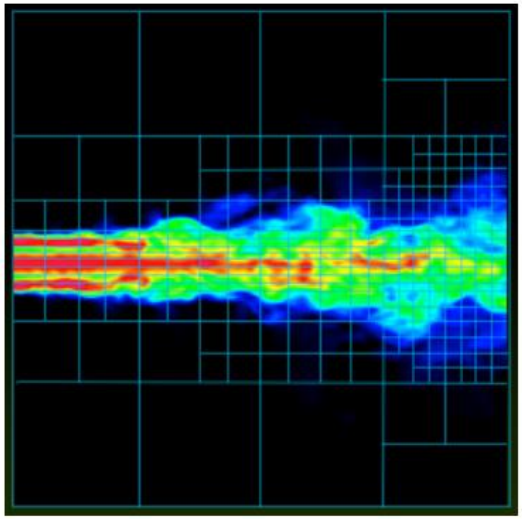
Higher performance,
lower accuracy

Fine grid



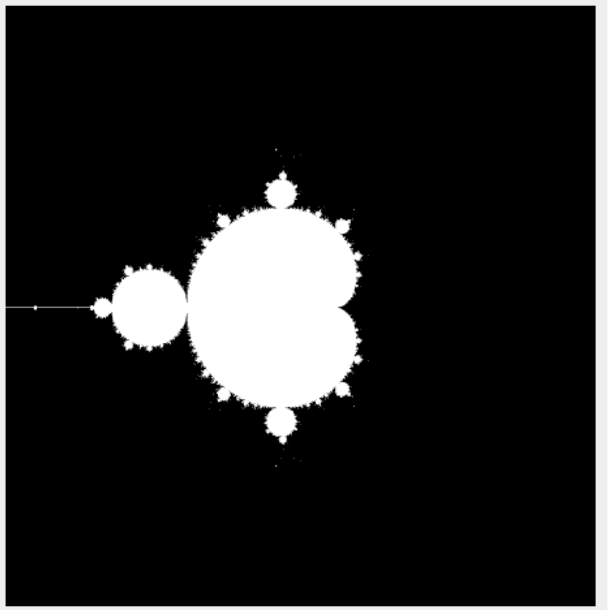
Lower performance,
higher accuracy

Dynamic grid

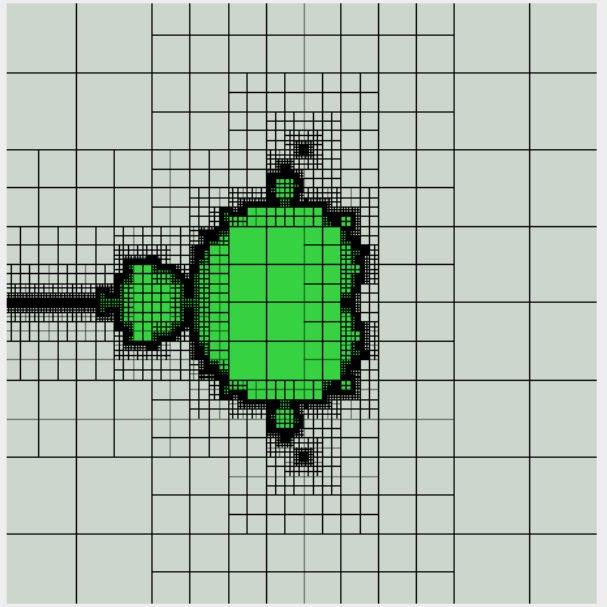


Target performance
where accuracy is required

Parallelism based on Level of Detail



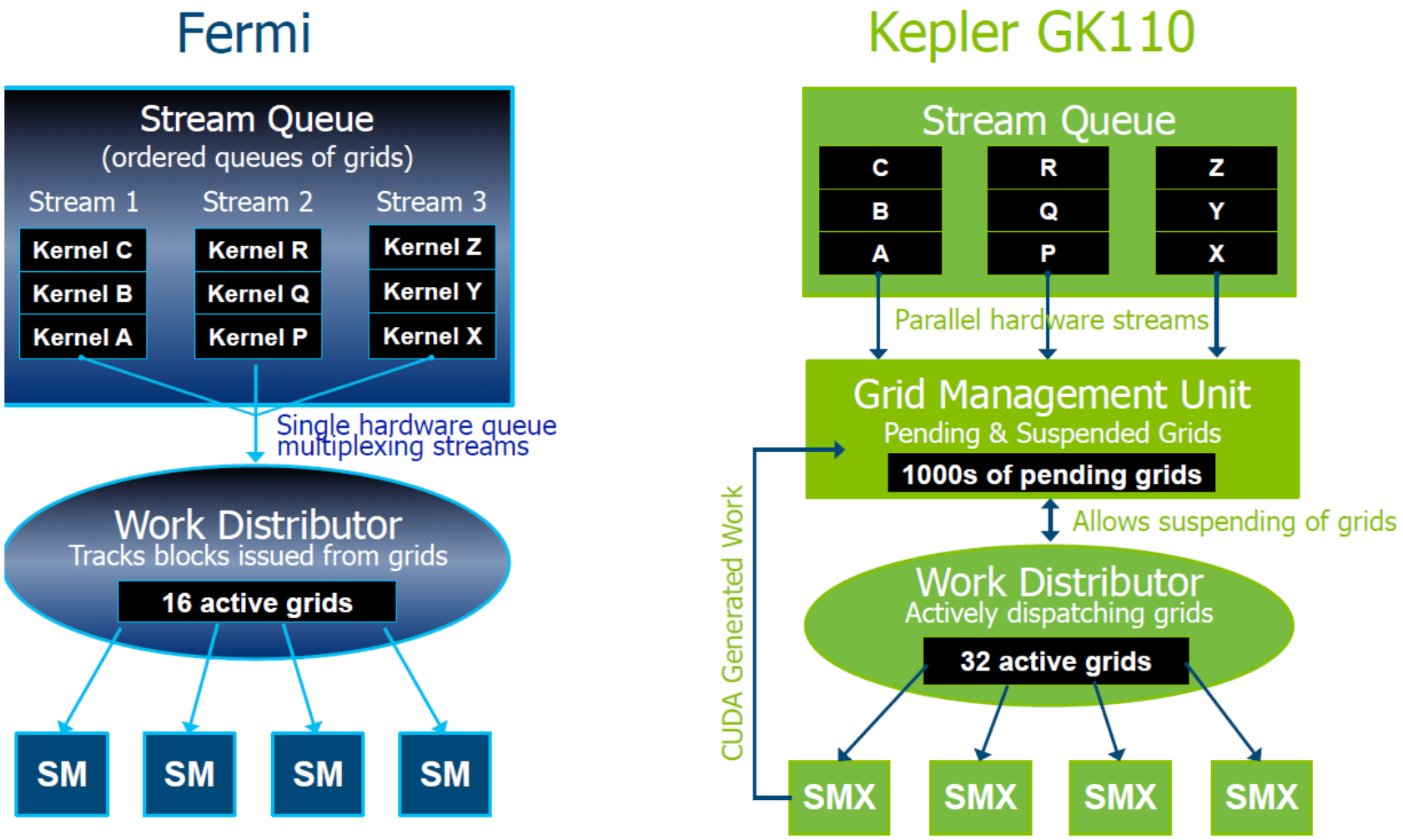
Computational power allocated to regions of interest



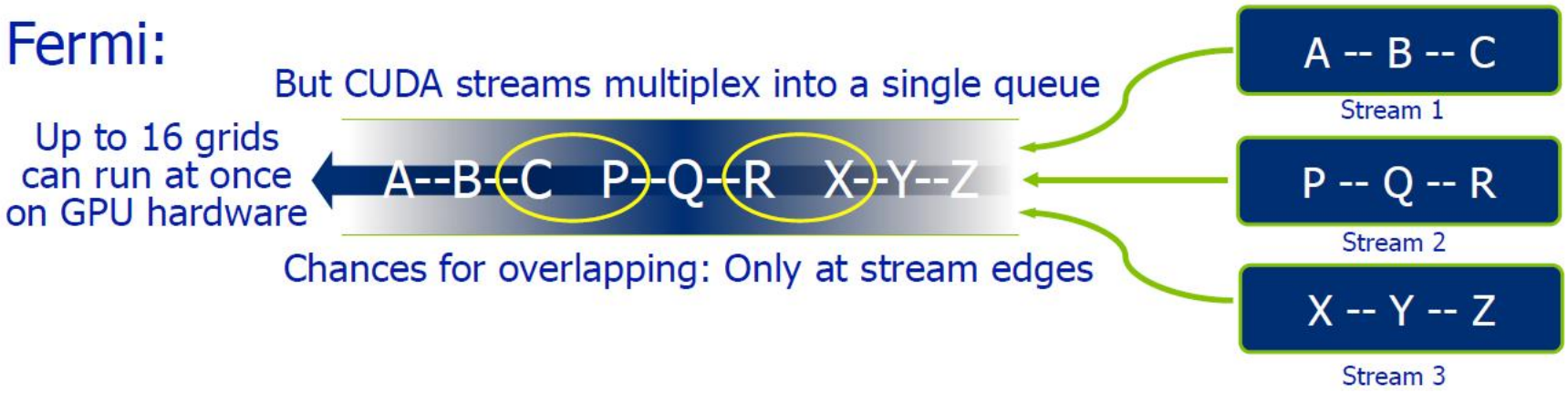
CUDA until 2012:
- The CPU launches kernels regularly.
- All pixels are treated the same.

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

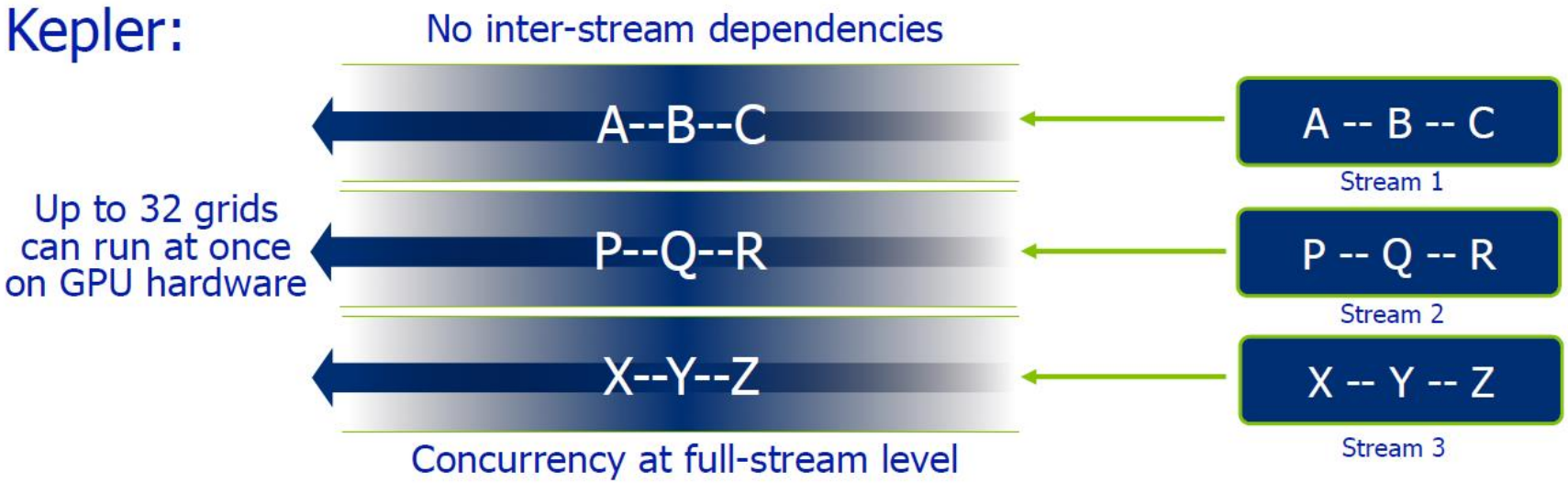
Grid Management Unit



Software and Hardware Queues



Software and Hardware Queues



Instruction Issue and Execution

	SM-SMX fetch & issue (front-end)	SM-SMX execution (back-end)
Fermi (GF100)	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.	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). A total of up to **4 concurrent warps**.
Kepler (GK110)	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.	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**.

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);
```

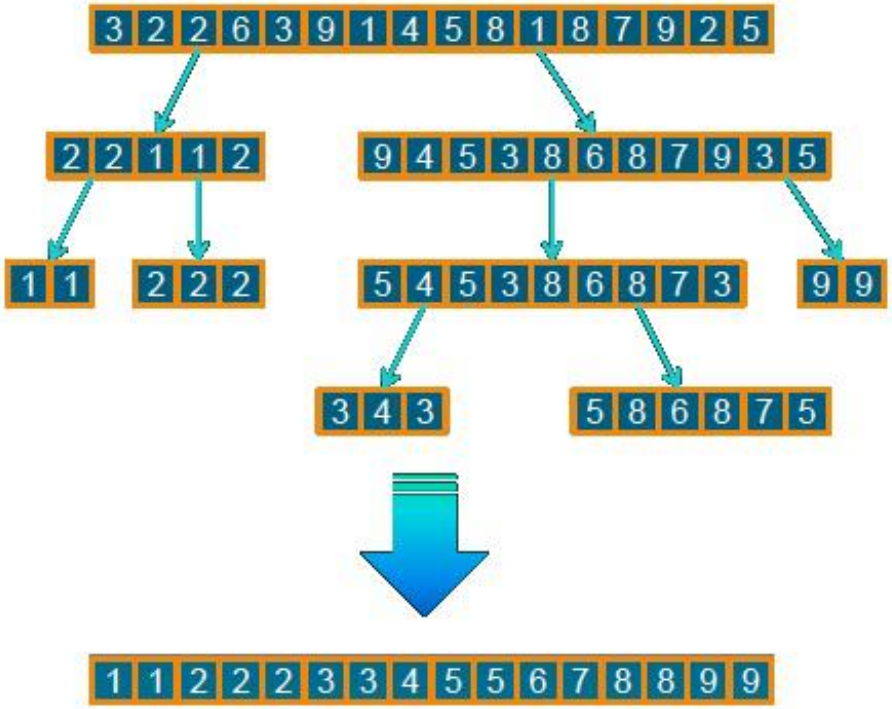
- Up to 24 nested loops supported in CUDA 5.0

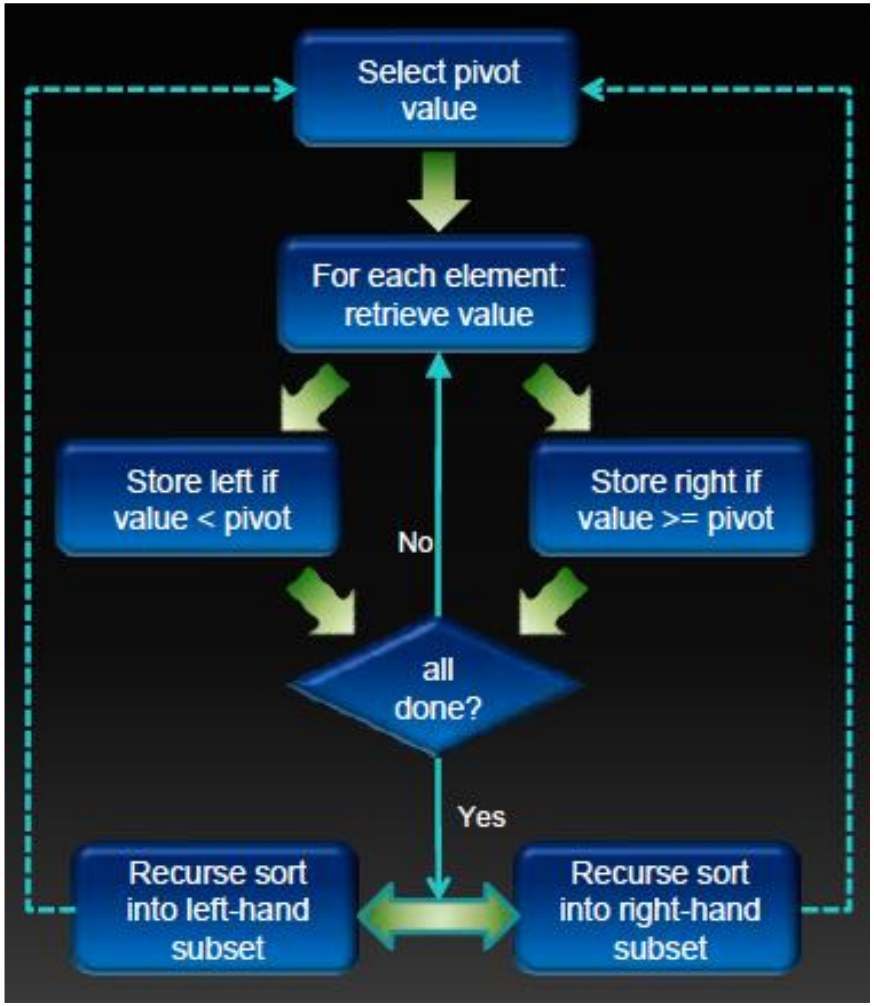
Recursive Parallel Algorithms prior to Kepler

- 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);
}
```

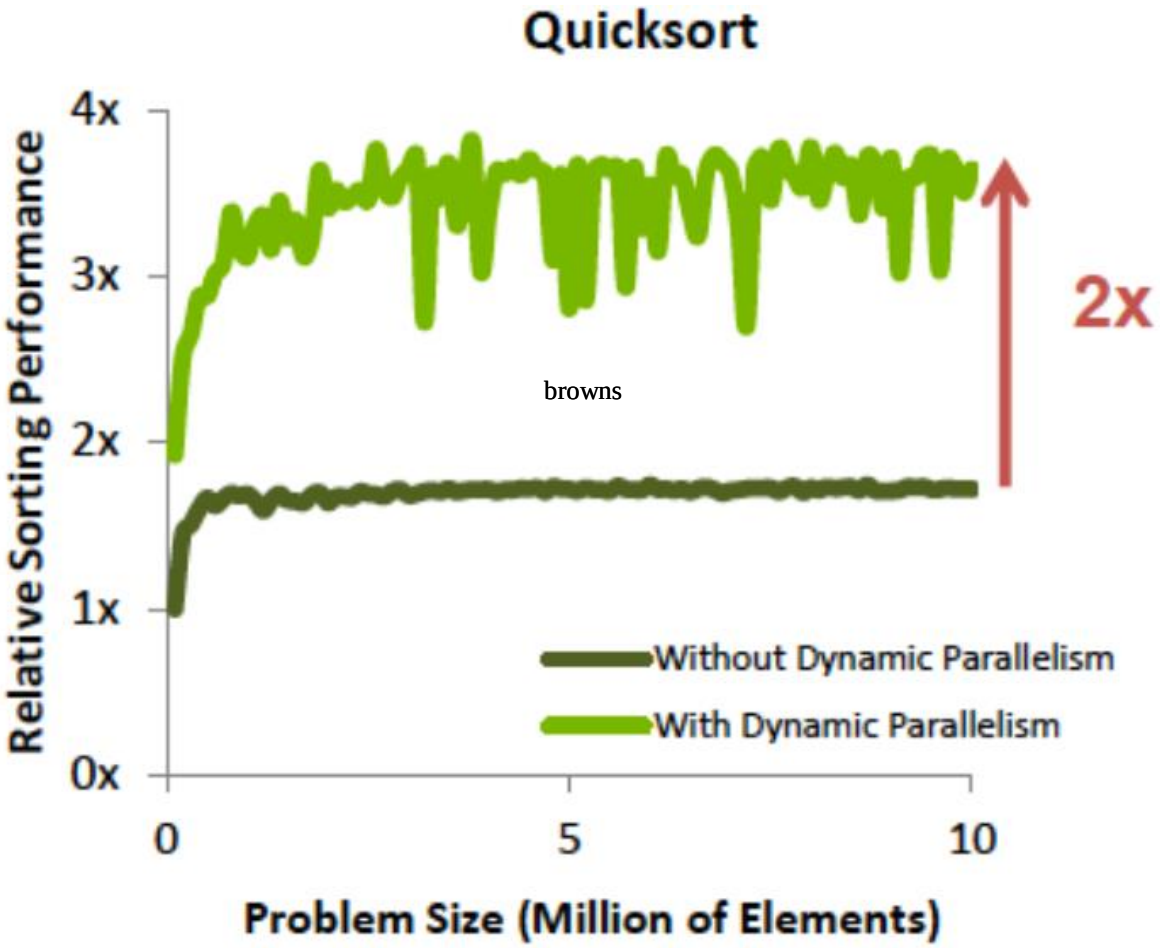
left- and right-hand sorts are serialized

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

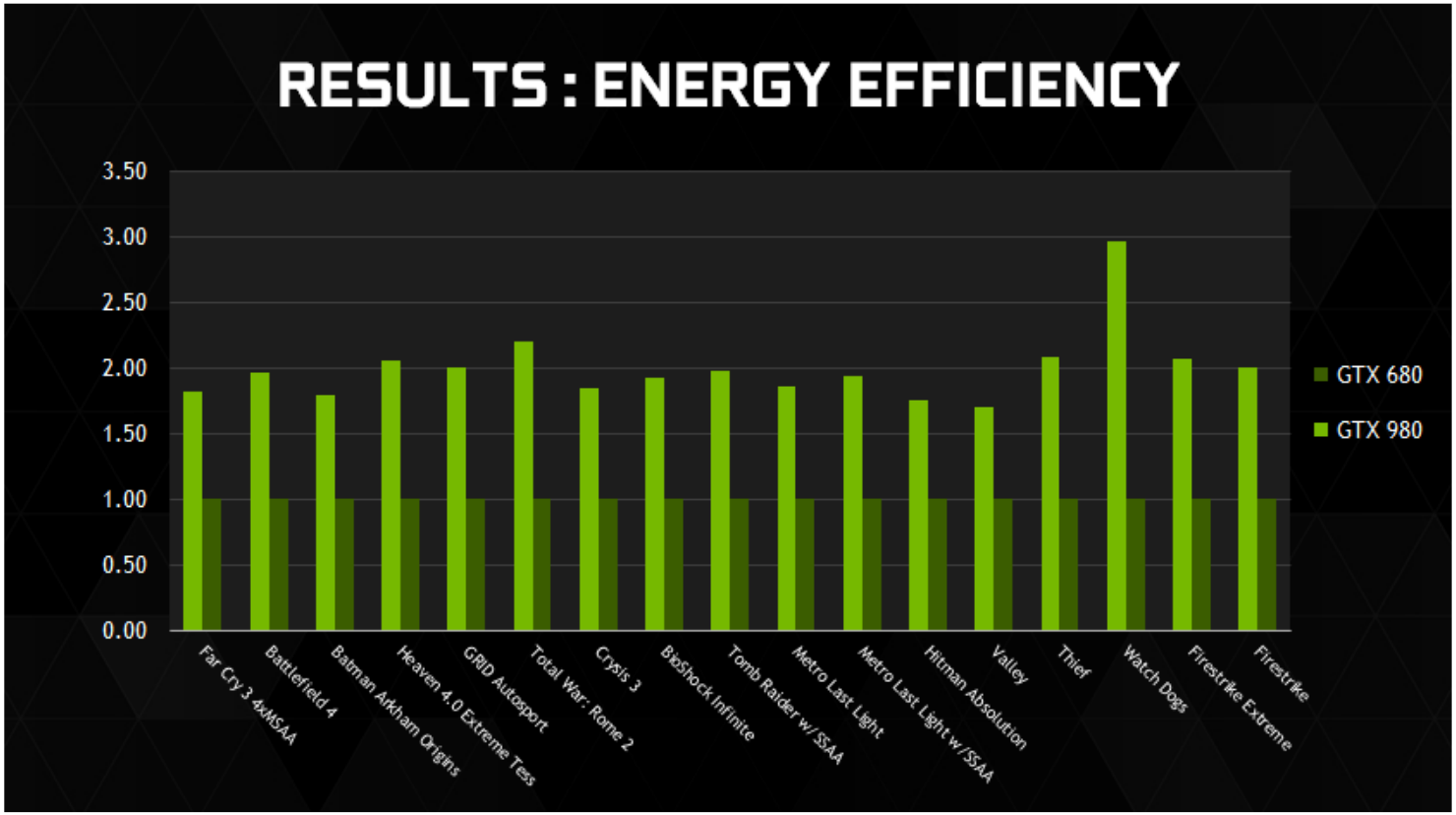


Maxwell
(2nd generation)

Released in 2014

Material by Mark Harris (NVIDIA)
and others

Energy Efficiency



Performance per Watt
GTX 680: Kepler GTX 980: Maxwell

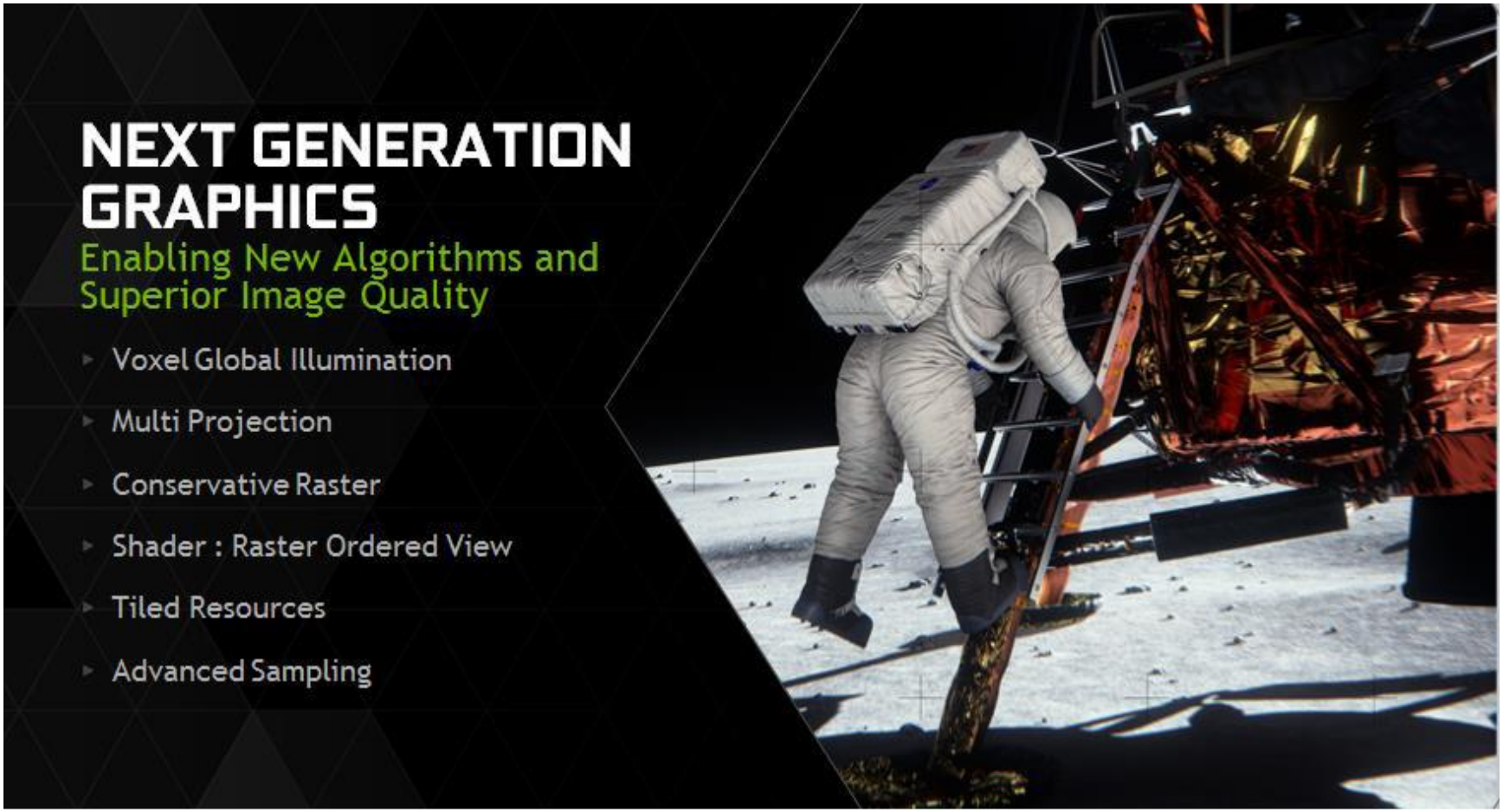
New Features

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, 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

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

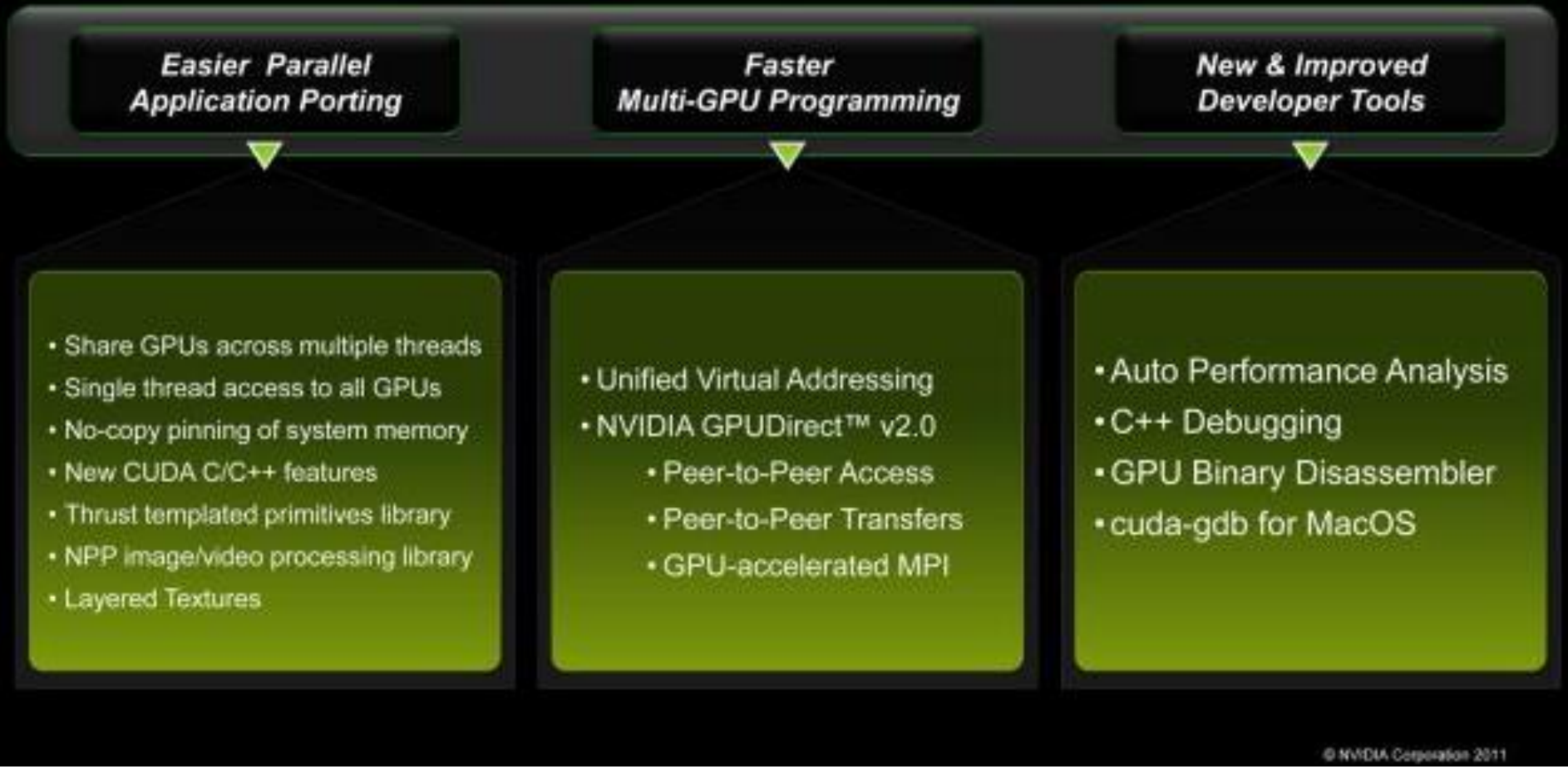
*Caffe Training on Multi-node Distributed-memory Systems Based on Intel® Xeon® Processor E5 Family (extrapolated)
Gennady Fedorov (Intel)'s picture Submitted by Gennady Fedorov (Intel), Vadim P. (Intel) on October 29, 2015
https://software.intel.com/en-us/articles/caffe-training-on-multi-node-distributed-memory-systems-based-on-intel-xeon-processor-e5

AMD RX Vega

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

CUDA 4.0

CUDA 4.0: Highlights



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

No-copy Pinning of System Memory

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

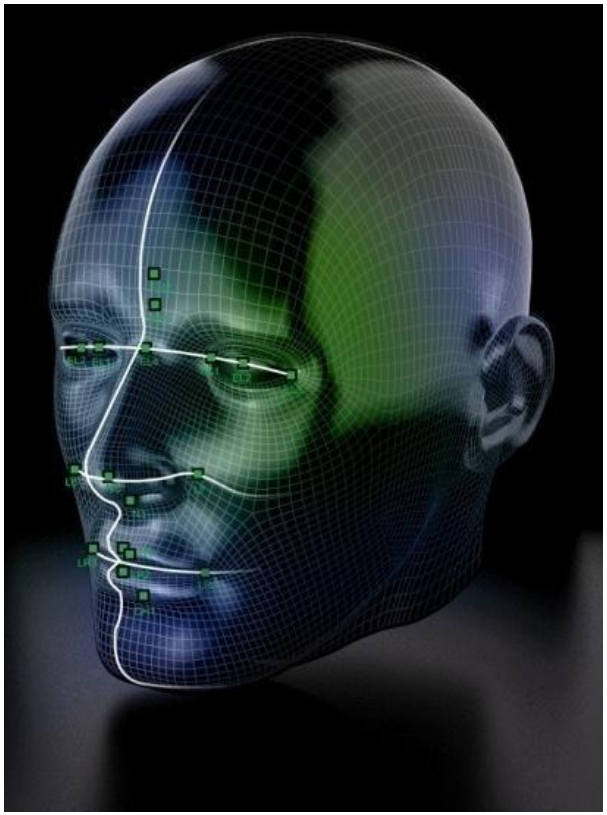
Before No-copy Pinning	With No-copy Pinning
Extra allocation and extra copy required	Just register and go!
malloc(a)	
cudaMallocHost(b)	
memcpy(b, a)	cudaHostRegister(a)
cudaMemcpy() to GPU, launch kernels, cudaMemcpy() from GPU	
memcpy(a, b)	
cudaFreeHost(b)	cudaHostUnregister(a)

New CUDA C/C++ Language Features

- C++ new/delete
  - Dynamic memory management
- C++ virtual functions
  - Easier porting of existing applications
- Inline PTX
  - Enables assembly-level optimization

GPU-Accelerated Image Processing

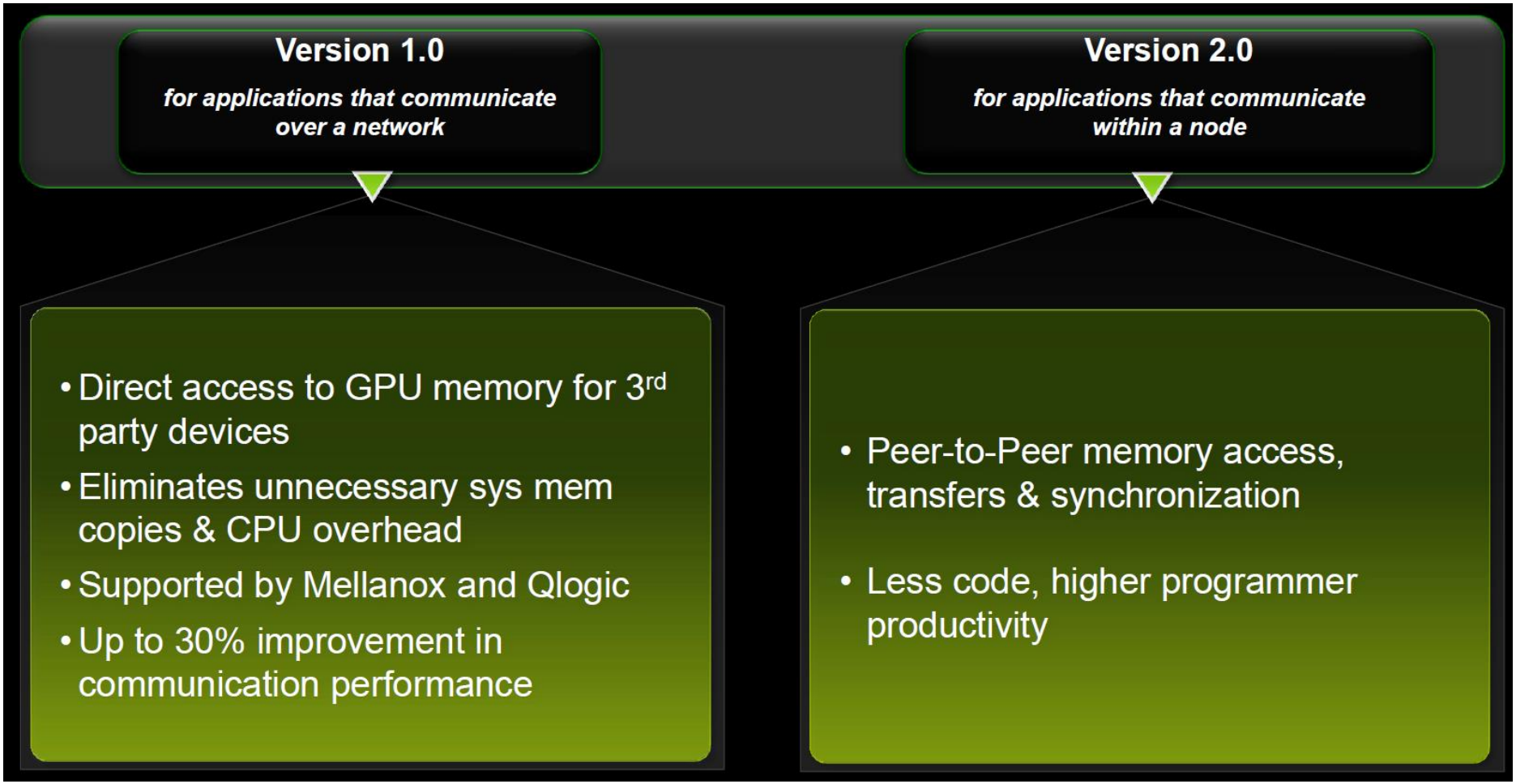
- NVIDIA Performance Primitives (NPP) library
  - 10x to 36x faster image processing
  - Initial focus on imaging and video related primitives
    - Data exchange and initialization
    - Color conversion
    - Threshold and compare operations
    - Statistics
    - Filter functions
    - Geometry transforms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



Layered Textures – Faster Image Processing

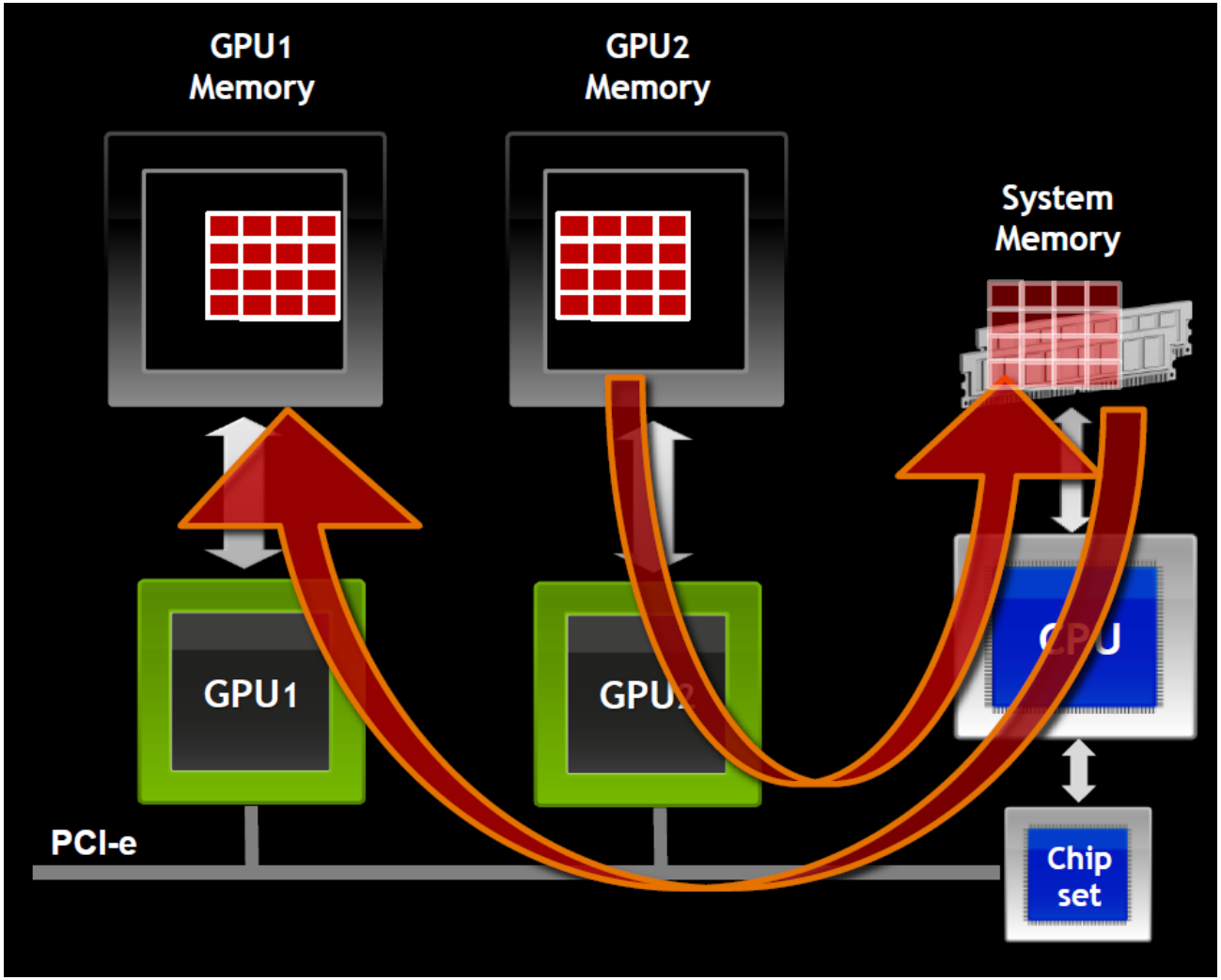
- Ideal for processing multiple textures with same size/format
  - Large sizes supported on Tesla T20 (Fermi) GPUs (up to 16k x 16k x 2k)
  - e.g. Medical Imaging, Terrain Rendering (flight simulators), etc.
- Faster Performance
  - Reduced CPU overhead: single binding for entire texture array
  - Faster than 3D Textures: more efficient filter caching
  - Fast interop with OpenGL / Direct3D for each layer
  - No need to create/manage a texture atlas
- No sampling artifacts
  - Linear/Bilinear filtering applied only within a layer

NVIDIA GPUDirect:Towards Eliminating the CPU Bottleneck



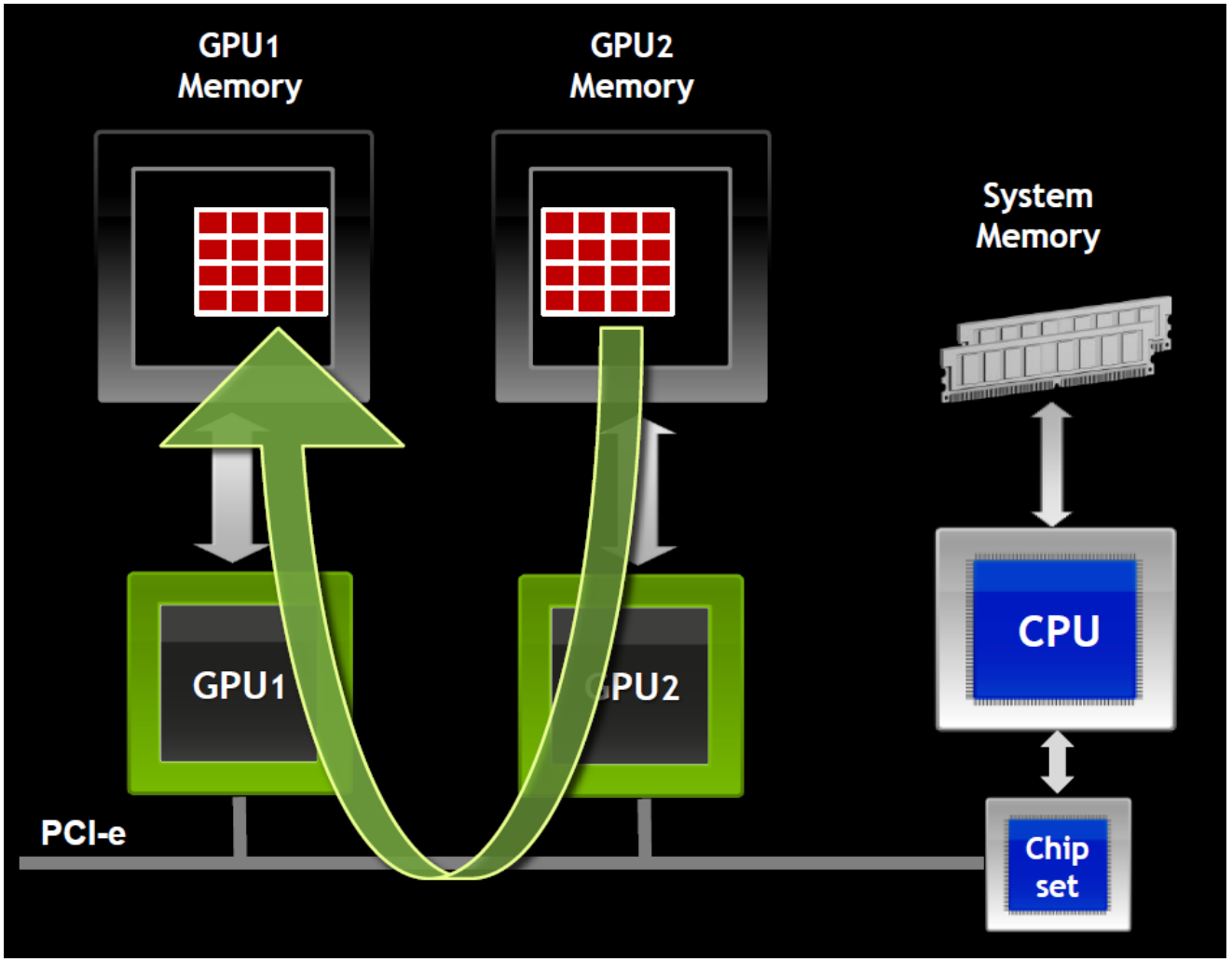
Before GPUDirect 2.0

Two copies required



GPUDirect 2.0: Peer-to-Peer Communication

Only one copy required

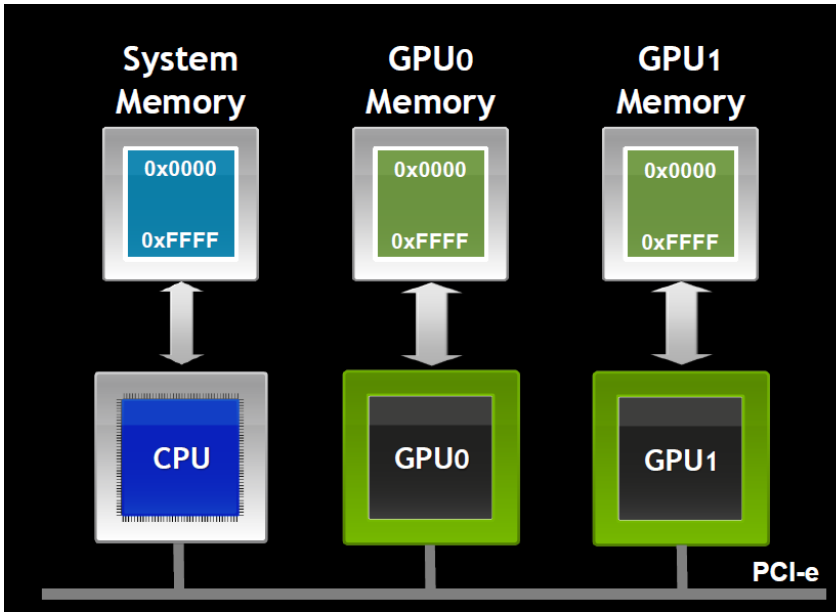


GPUDirect 2.0: Peer-to-Peer Communication

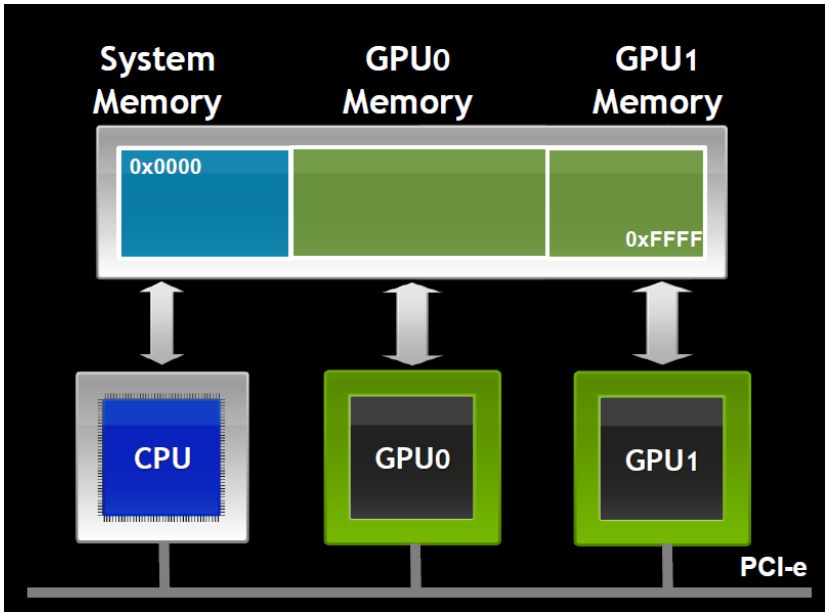
- Direct communication between GPUs
  - Faster - no system memory copy overhead
  - More convenient multi-GPU programming
- Direct Transfers
  - Copy from GPU0 memory to GPU1 memory
  - Works transparently with UVA
- Direct Access
  - GPU0 reads or writes GPU1 memory (load/store)
- Supported on Tesla 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
cudaMemcpyHostToHost cudaMemcpyHostToDevice cudaMemcpyDeviceToHost cudaMemcpyDeviceToDevice	cudaMemcpyDefault (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

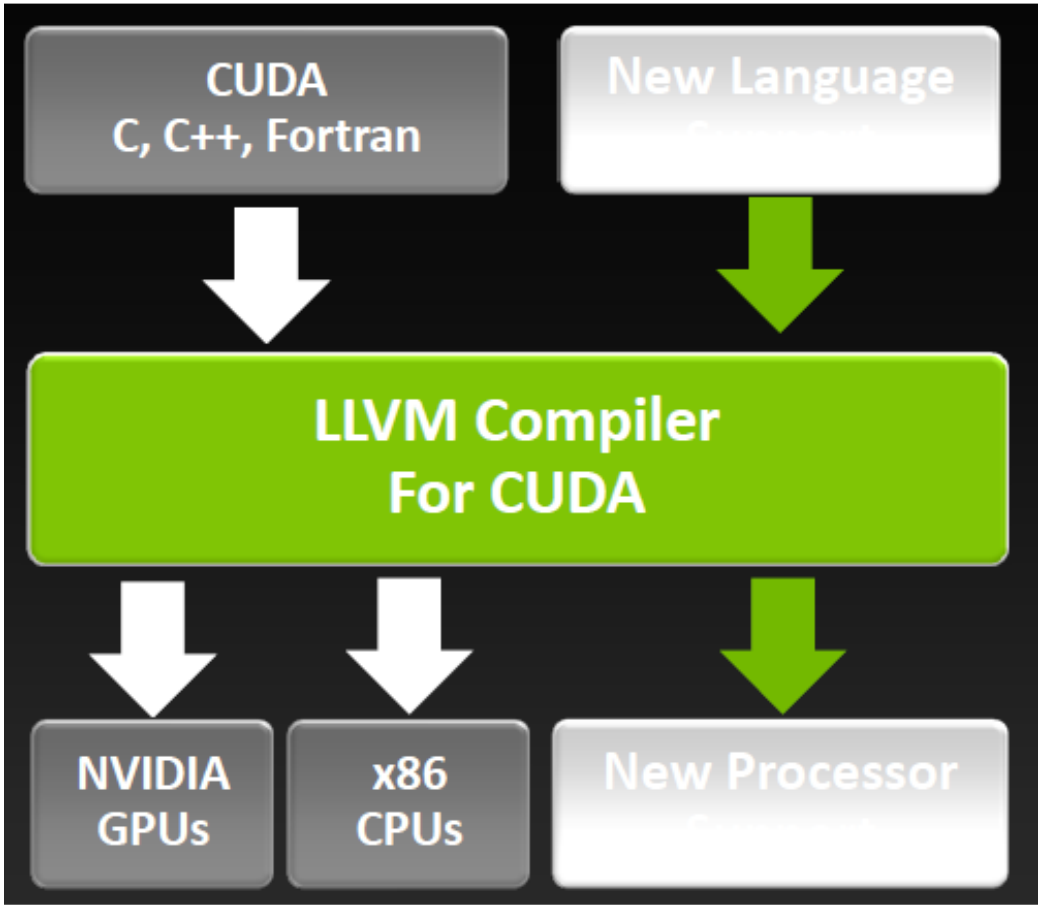
CUDA 5.0

Mark Harris

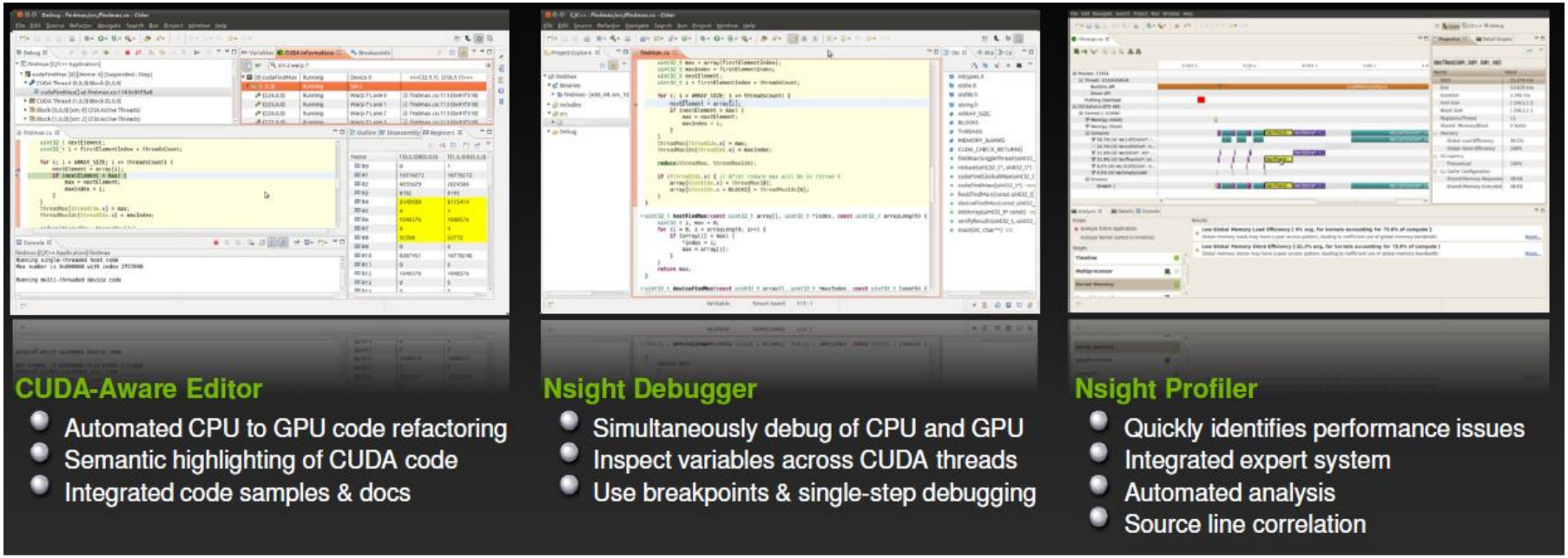
Chief Technologist, GPU Computing

Open Source LLVM Compiler

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

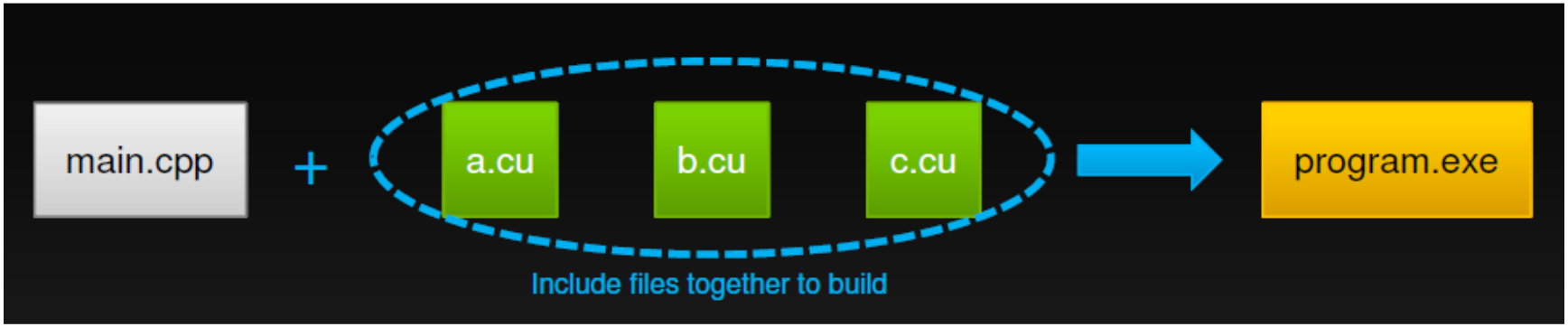


NVIDIA Nsight, Eclipse Edition



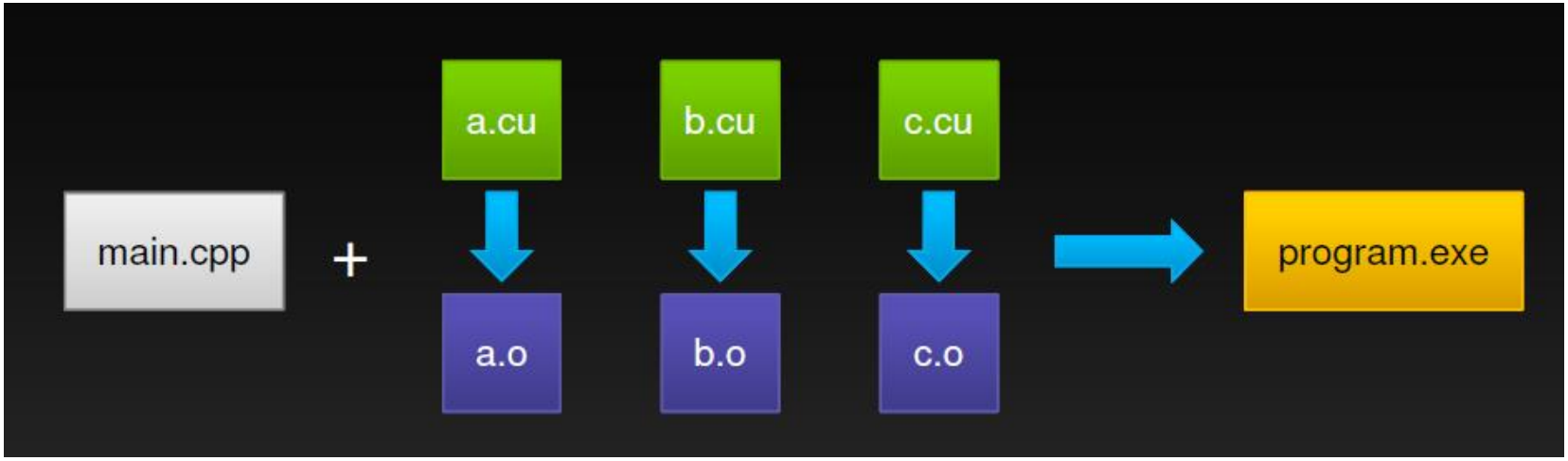
For Linux and Mac OS

CUDA 4: Whole-Program Compilation & Linking



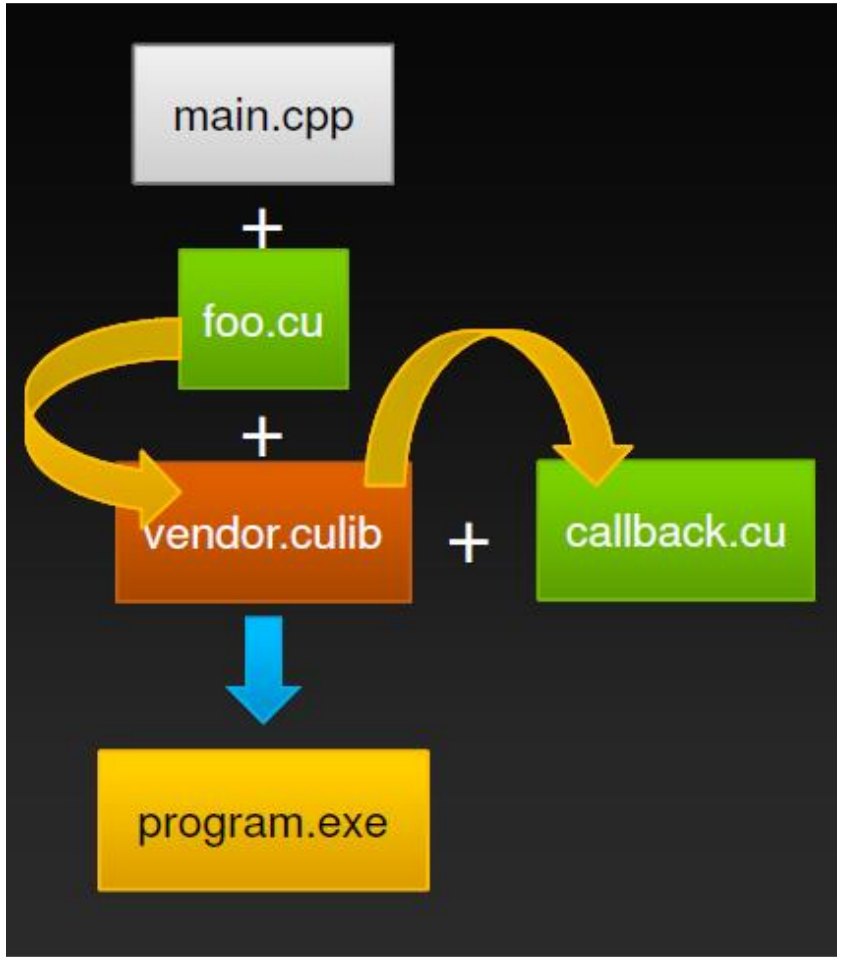
CUDA 5: GPU Library Object Linking

- 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)

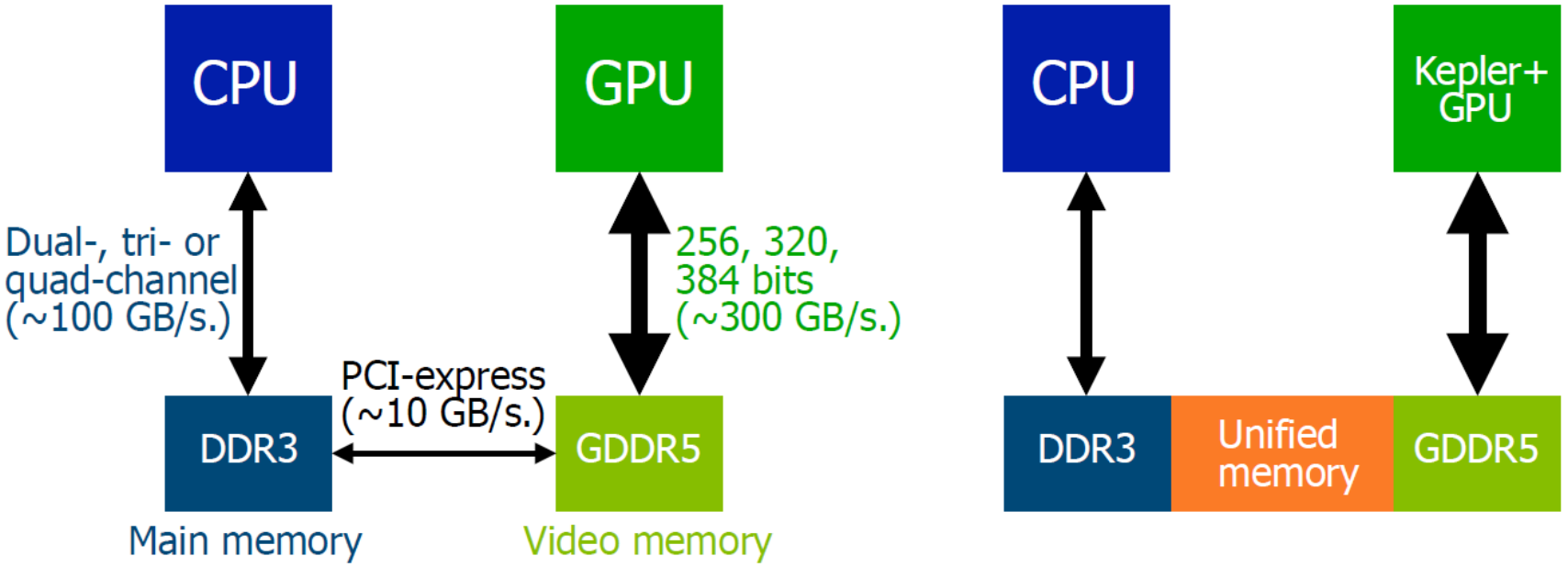
CUDA 6 Highlights

- 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

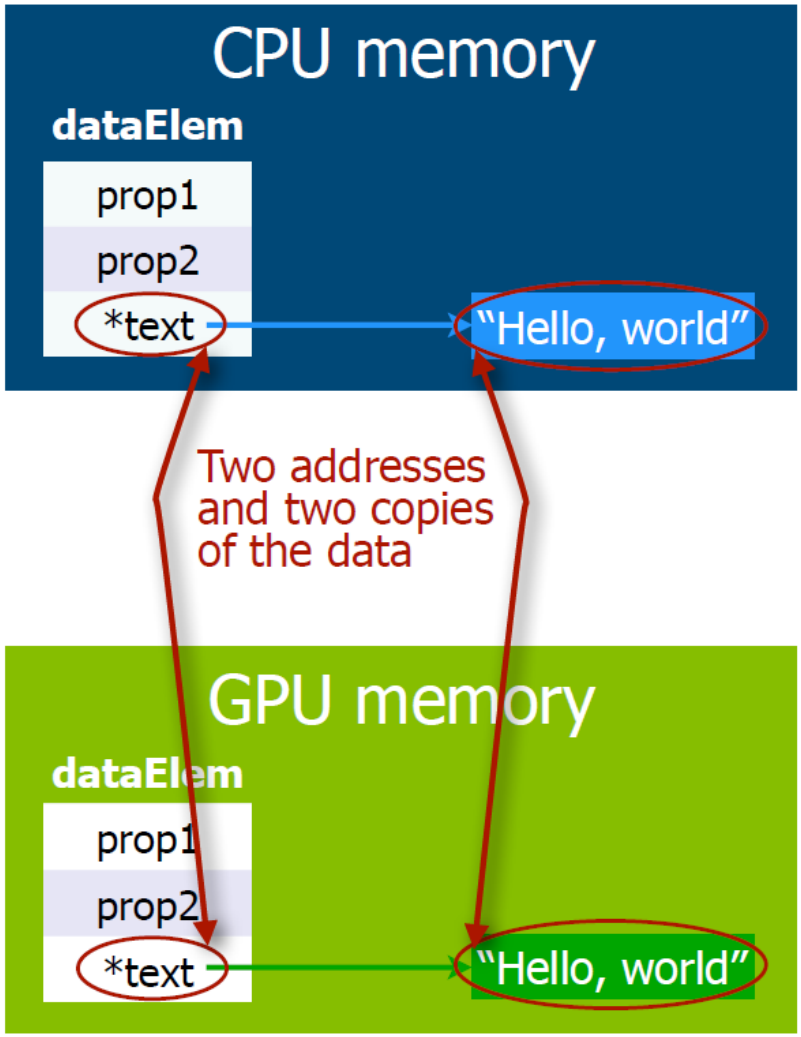
Memory Types

	Zero-Copy (pinned memory)	Unified Virtual Addressing	Unified Memory
CUDA call	cudaMallocHost(&A, 4);	cudaMalloc(&A, 4);	cudaMallocManaged(&A, 4);
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

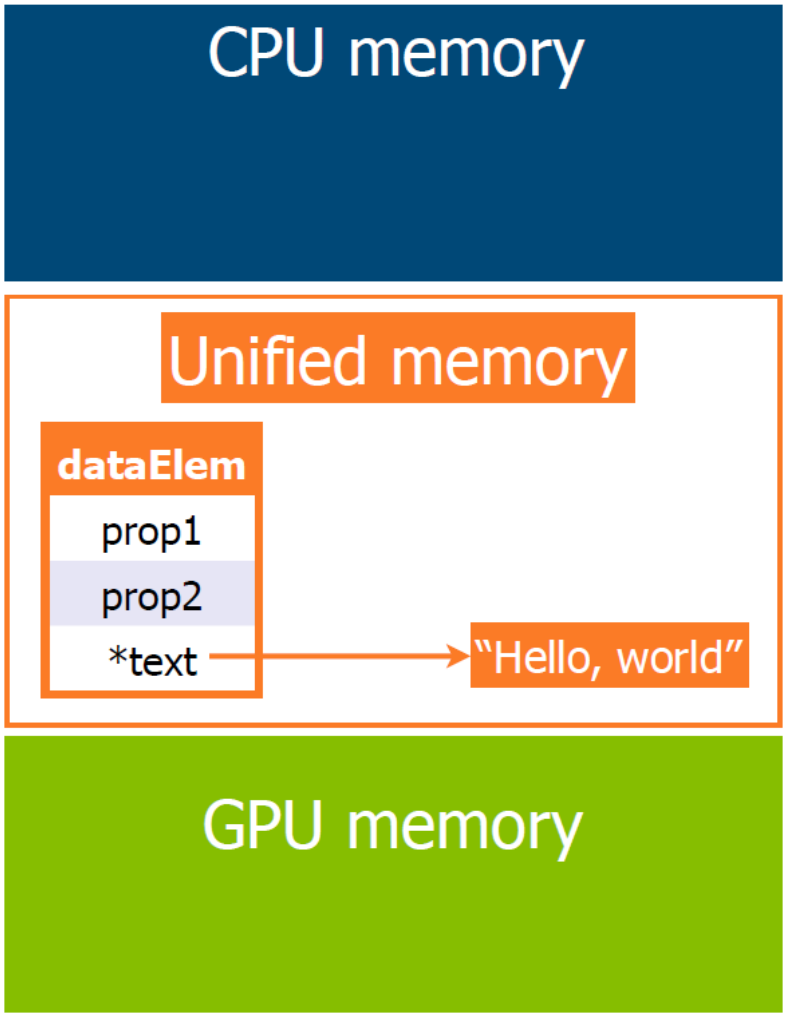
- New call: **cudaMallocManaged()**
  - Drop-in replacement for cudaMalloc() allocates managed memory
  - Returns pointer accessible from both Host and Device
- New call: **cudaStreamAttachMemAsync()**
  - Manages concurrency in multi-threaded CPU applications
- New keyword: **__managed__**
  - Declares 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




```
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

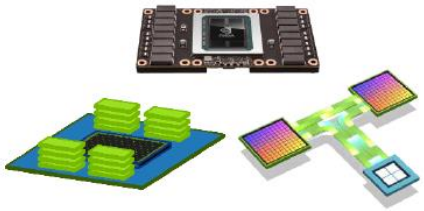
CUDA 8.0

By Milind Kukanur
NVIDIA

What's New

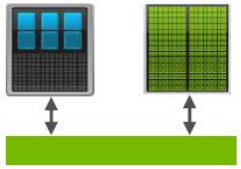
PASCAL SUPPORT

New Architecture
NVLINK
HBM2 Stacked Memory
Page Migration Engine



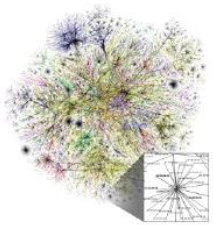
UNIFIED MEMORY

Larger Datasets
Demand Paging
New Tuning APIs
Data Coherence & Atomics



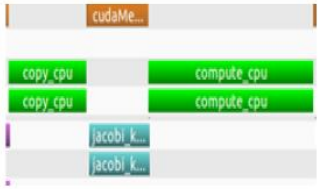
LIBRARIES

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);
}
```