

Inside Kepler

Manuel Ujaldon

Nvidia CUDA Fellow

Computer Architecture Department
University of Malaga (Spain)

Talk outline [46 slides]

1. Introducing the architecture [2]
2. Cores organization [9]
3. Memory and data transport [6]
4. Major software and hardware enhancements [8]
 1. Software: Relaxing constraints on massive parallelism.
 2. Hardware: Grid dimensions, dynamic parallelism and Hyper-Q.
5. Exploiting on Kepler the new capabilities [21]
 1. Dynamic load balancing [2].
 2. Thread scheduling [8].
 3. Data-dependent execution [2].
 4. Recursive parallel algorithms [4].
 5. Library calls from kernels [3].
 6. Simplify CPU/GPU division [2].

1. Introducing the architecture

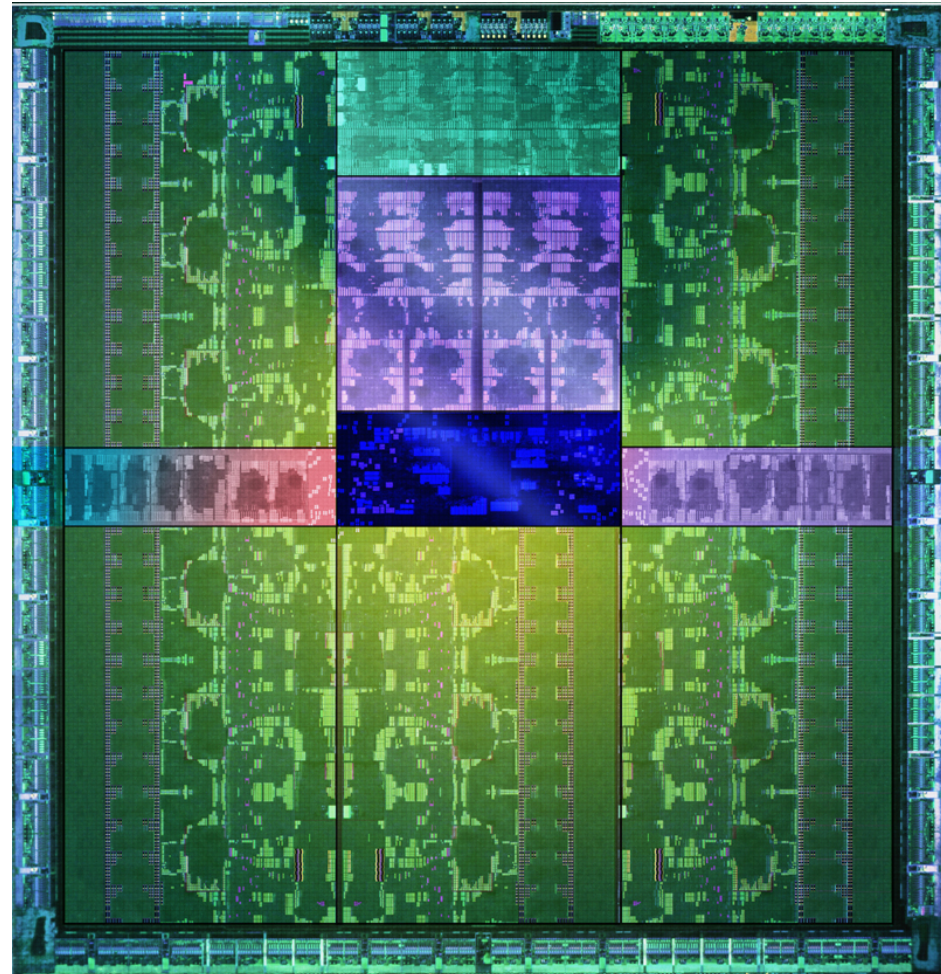


The three pillars of Kepler

Power consumption

Performance

Programmability



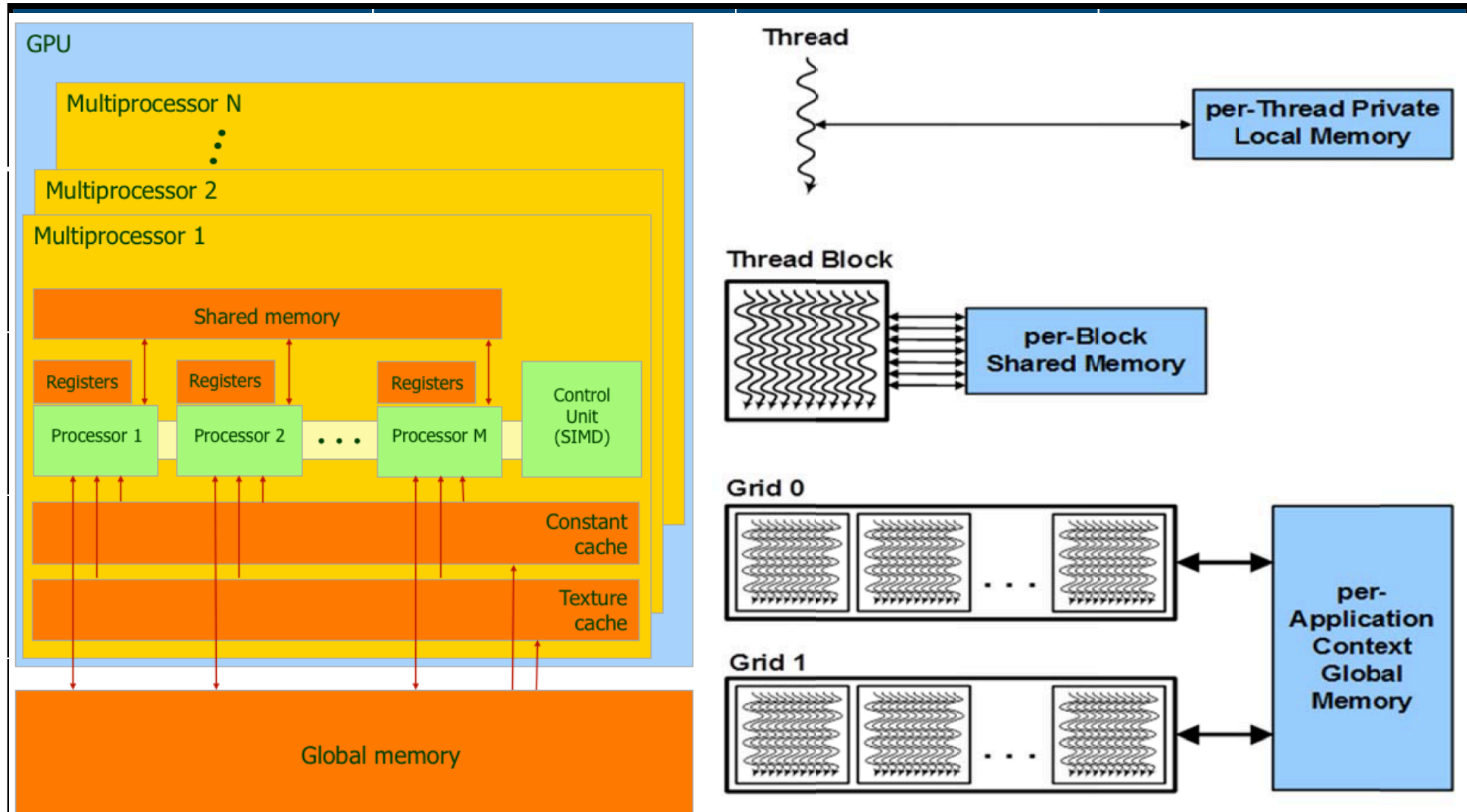
Summary of the most outstanding features

- **Manufacturing:** 7100 million trans. @ 28 nm. by TSMC.
- **Architecture:** Between 7 and 15 multiprocessors SMX, endowed with 192 cores each.
 - The number of multiprocessors depends of the GK version [GKxxx].
- **Arithmetic:** More than 1 TeraFLOP in double precision (64 bits IEEE-754 floating-pointing format).
 - Specific values depend on the clock frequency for each model (usually, more on GeForces, less on Teslas).
 - We can reach 1 PetaFLOPS with only 10 server racks.
- **Major innovations in core design:**
 - Dynamic parallelism.
 - Thread scheduling (Hyper-Q).

2. Cores organization



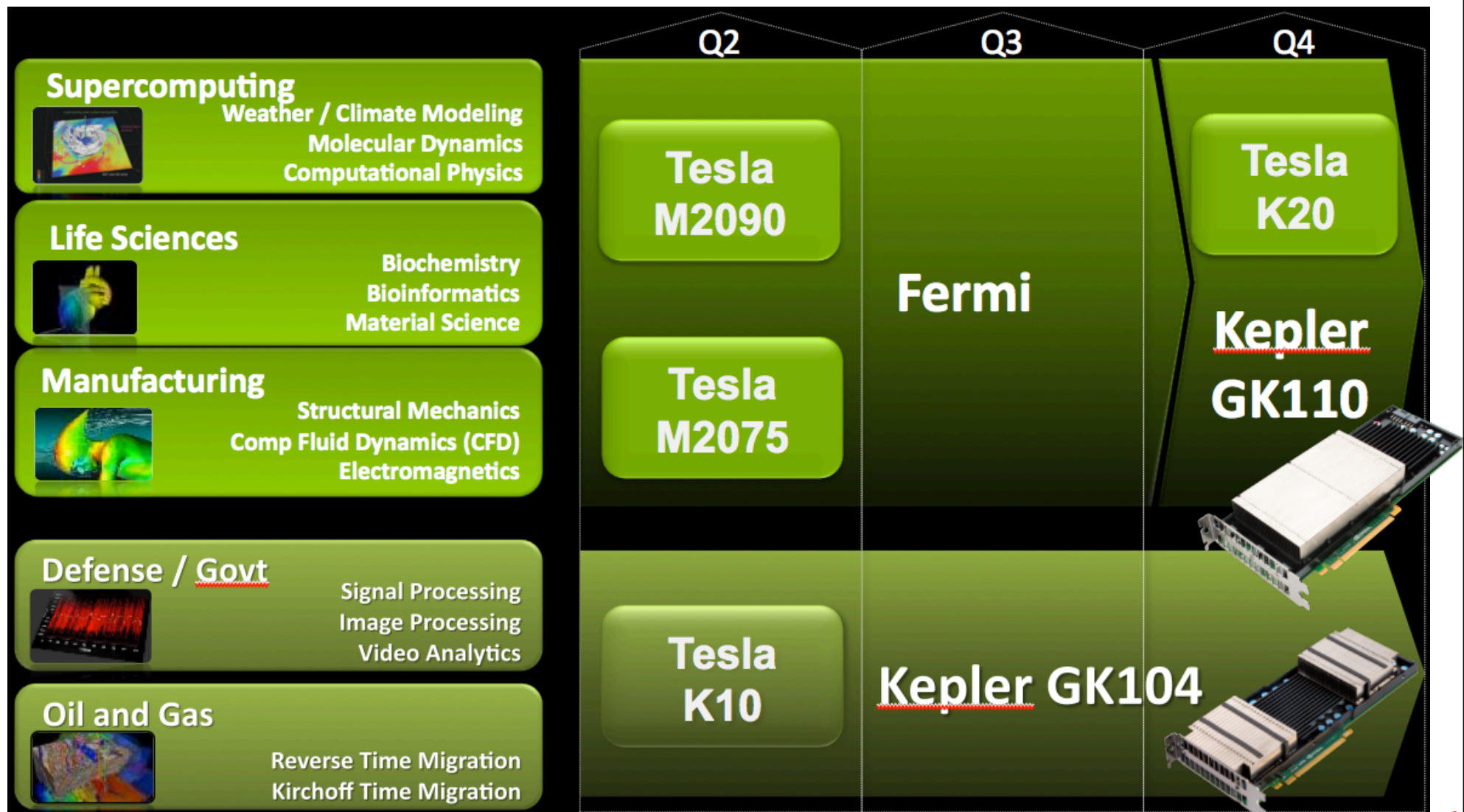
A brief reminder of what CUDA is about



... and 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 |

High-end, mid-end and low-end cards: Applications and time frame (2012)



Kepler in perspective:

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 |

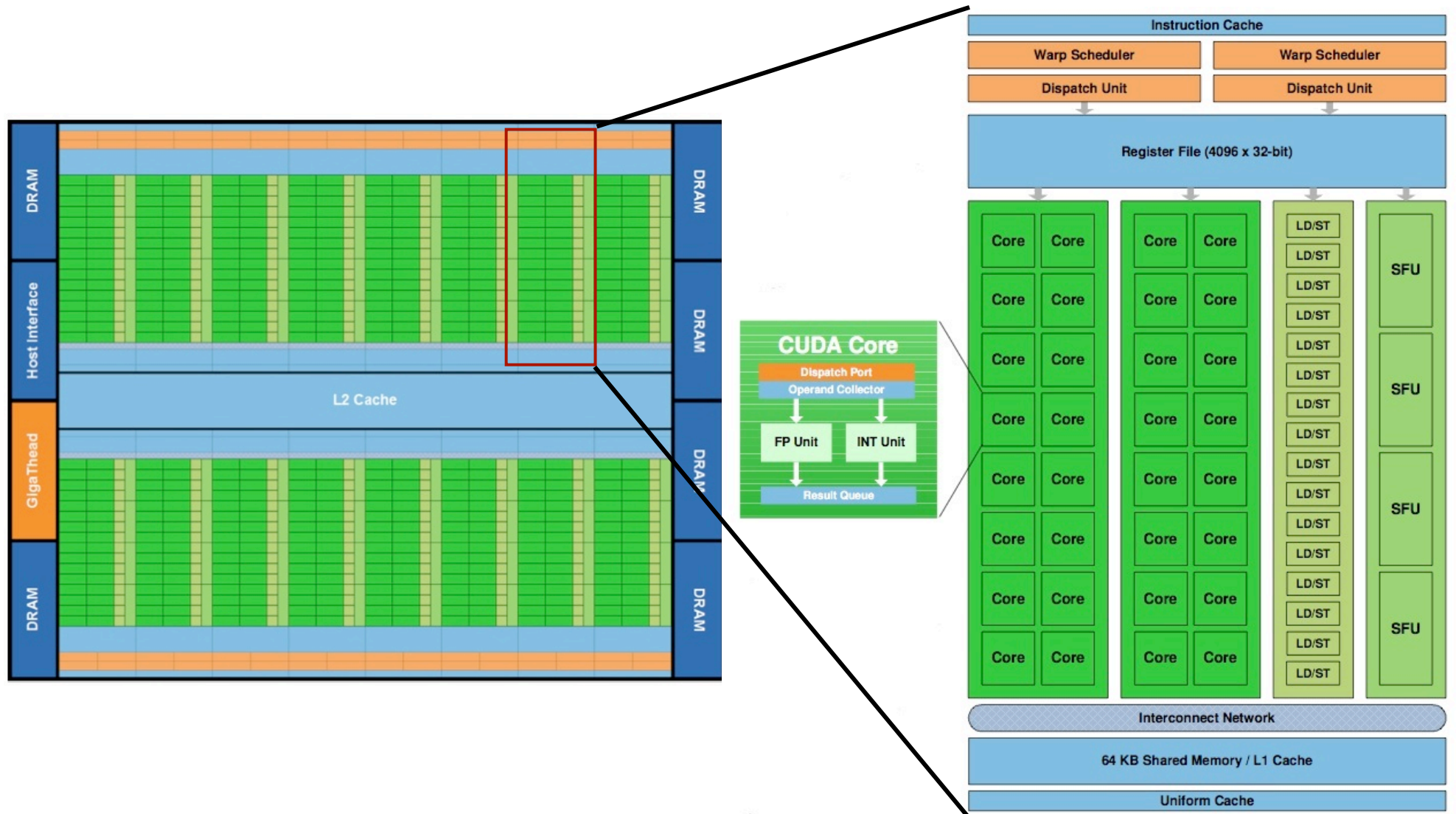
Kepler in perspective: Power consumption

| Tesla card | M2075 | M2090 | K10 | K20 | K20X |
|------------------------------------|----------|----------|-----------|---------|---------|
| Total number of cores | 448 | 512 | 1536 (x2) | 2496 | 2688 |
| Core clock frequency | 1150 MHz | 1300 MHz | 745 MHz | 706 MHz | 732 MHz |
| Thermal design power | 225 W | 225 W | 225 W | 225 W | 235 W |
| Number of single precision cores | 448 | 512 | 1536 (x2) | 2496 | 2688 |
| GFLOPS (peak single precision) | 1030 | 1331 | 2288 (x2) | 3520 | 3950 |
| GFLOPS per watt (single precision) | 4.17 | 4.75 | 20.35 | 15.64 | 16.71 |
| Number of double precision cores | 224 | 256 | 64 (x2) | 832 | 896 |
| GFLOPS (peak double precision) | 515 | 665 | 95 (x2) | 1170 | 1310 |
| GFLOPS per watt (double precision) | 2.08 | 2.37 | 0.85 | 5.21 | 5.57 |

Kepler in perspective: Memory features

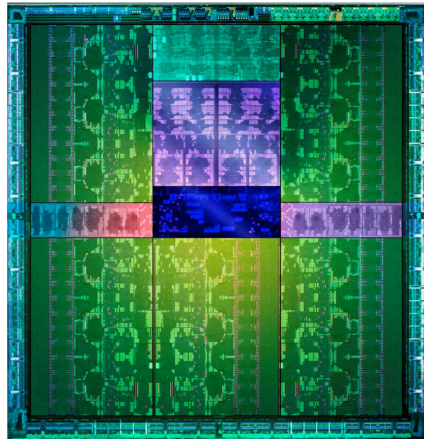
| Tesla card | M2075 | M2090 | K10 | K20 | K20X |
|---------------------------------------|-------------|-------------|------------|------------|------------|
| 32-bit register file / multiprocessor | 32768 | 32768 | 65536 | 65536 | 65536 |
| L1 cache + shared memory size | 64 KB. | 64 KB. | 64 KB. | 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 |

Its predecessor Fermi

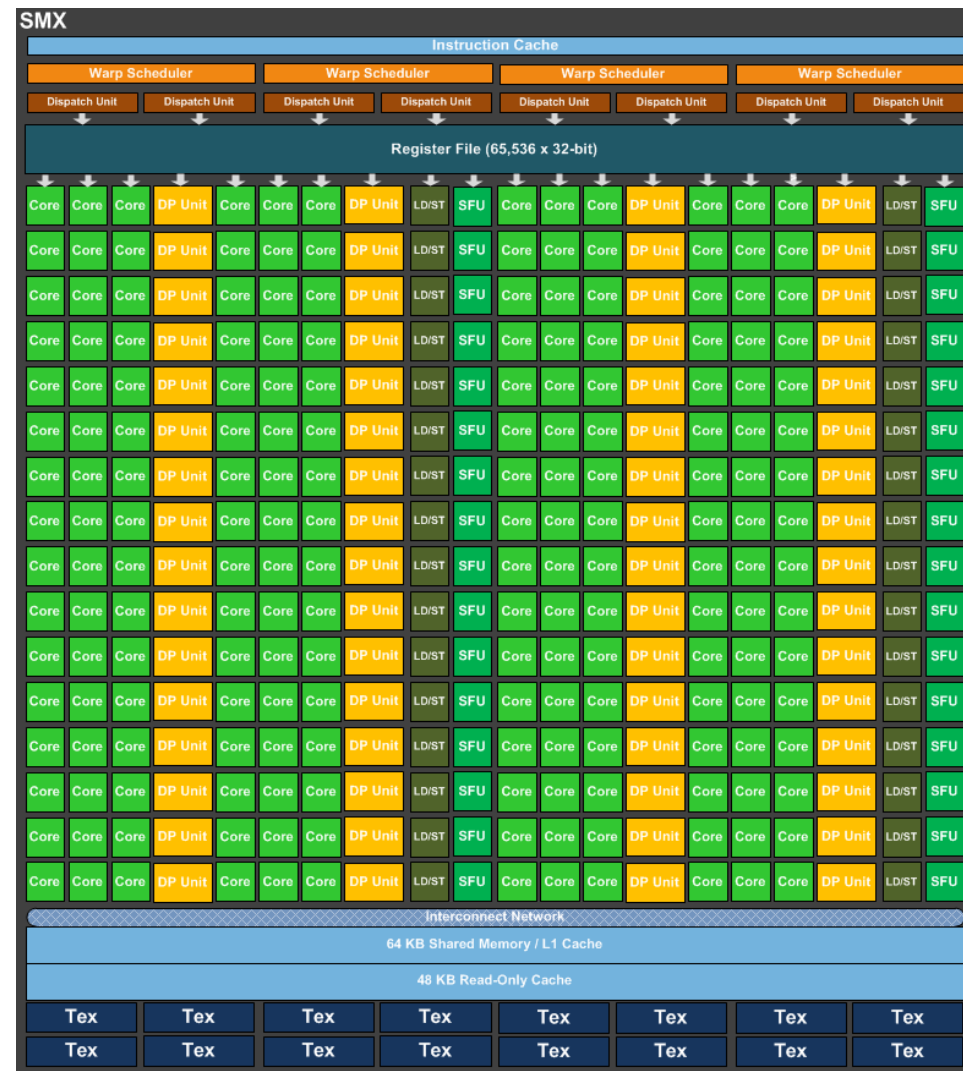
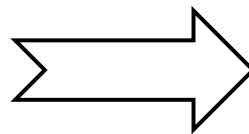
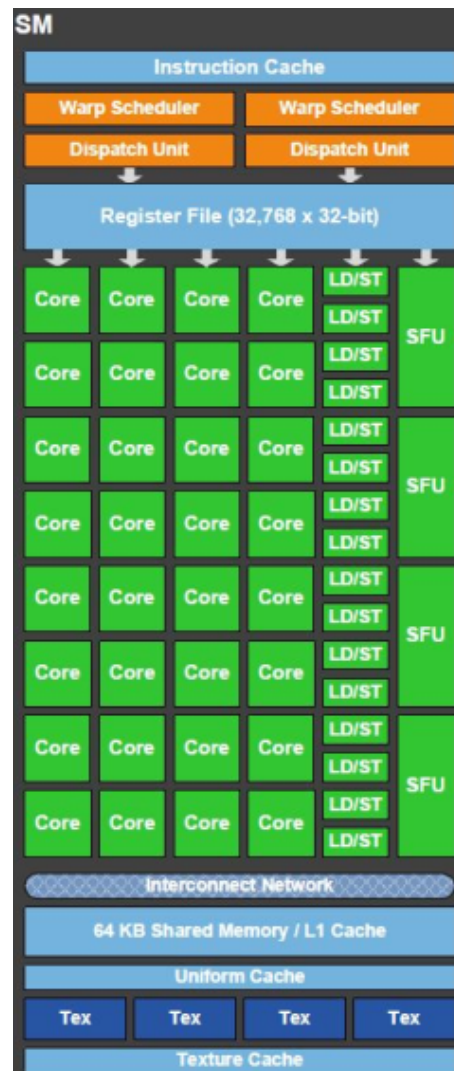


Kepler GK110:

Physical layout of functional units



From SM multiprocessor in Fermi GF100 to multiprocessor SMX in Kepler GK110



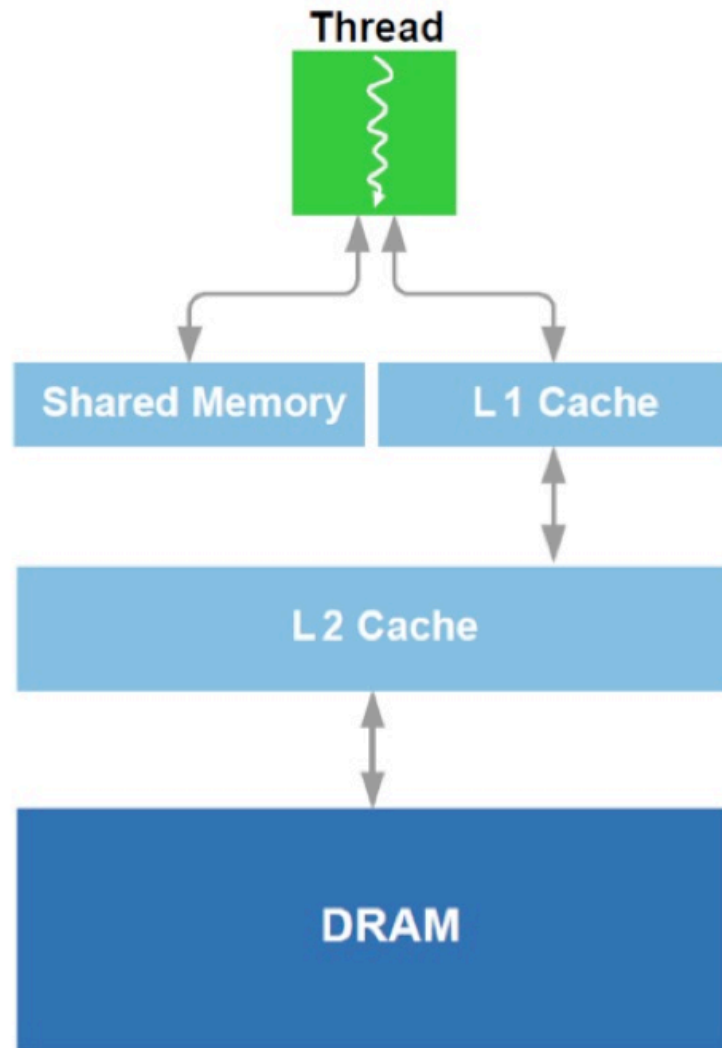
3. Memory and data transport



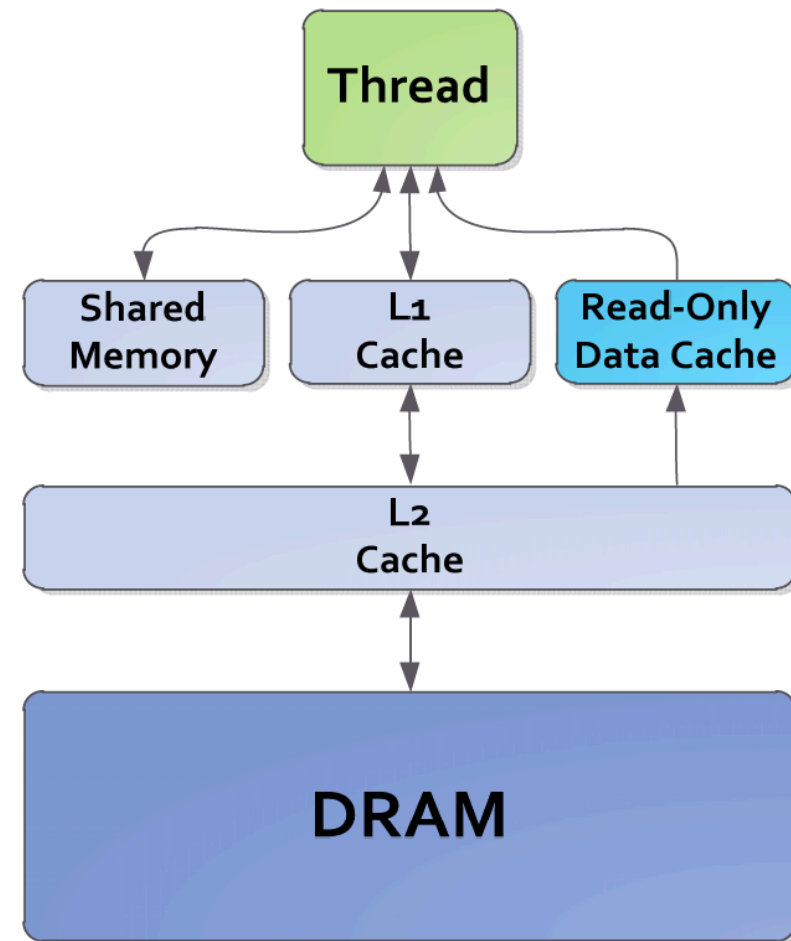
Enhancements in memory and data transport

- **Integrated memory** on each SMX. Versus Fermi's SM multiprocessors, Kepler duplicates:
 - The size and bandwidth for the register file.
 - The bandwidth for the shared memory.
 - The size and bandwidth for the L1 cache memory.
- **Internal memory (L2 cache):** 1.5 Mbytes.
- **External memory (DRAM):** GDDR5 and 384-bits for the data path (frequency and size depend on the graphics card).
- **Interface with the host:**
 - PCI-express v. 3.0 (actual bandwidth depends on motherboard).
 - Closer dialogs among video memories belonging to different GPUs.

Differences in memory hierarchy: Fermi vs. Kepler



Kepler Memory Hierarchy



Motivation for using the new data cache

- Additional 48 Kbytes to expand L1 cache size.
- Highest miss bandwidth.
- 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 the new 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;  
}
```

The memory hierarchy in numbers

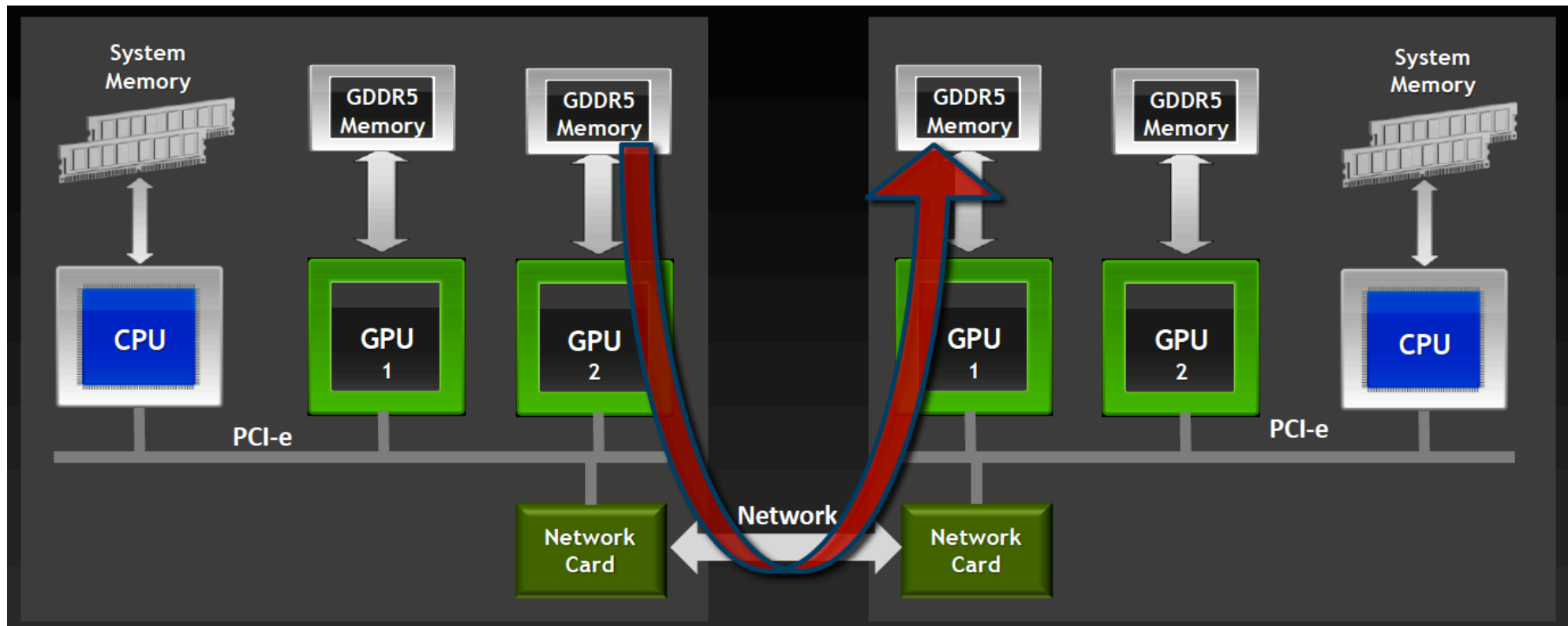
| GPU generation | Fermi | | Kepler | | Limitation | Impact |
|------------------------------------|---------|---------|------------|-----------------|------------|--------------|
| Hardware model | GF100 | GF104 | GK104 | GK110 | | |
| CUDA Compute Capability (CCC) | 2.0 | 2.1 | 3.0 | 3.5 | | |
| Max. 32 bits registers / thread | 63 | 63 | 63 | 255 | SW. | Working set |
| 32 bits registers / Multiprocessor | 32 K | 32 K | 64 K | 64 K | HW. | Working set |
| Shared memory / Multiprocessor | 16-48KB | 16-48KB | 16-32-48KB | 16-32-48 KB | HW. | Tile size |
| L1 cache / Multiprocessor | 48-16KB | 48-16KB | 48-32-16KB | 48-32-16 KB | HW. | Access speed |
| L2 cache / GPU | 768 KB. | 768 KB. | 768 KB. | 1536 KB. | HW. | Access speed |

● All Fermi and Kepler models are endowed with:

- ECC (Error Correction Code) in the video memory controller.
- Address bus 64 bits wide.
- Data bus 64 bits wide for each memory controller (few models include 4 controllers for 256 bits, most have 6 controllers for 384 bits)

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.



4. Major software and hardware enhancements



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 (on huge problem sizes):

| 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:

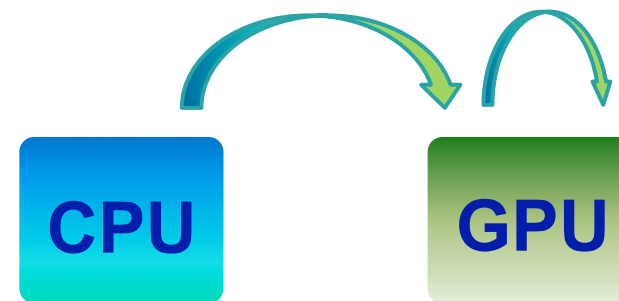
| GPU generation | Fermi | | Kepler | | Limitation | Impact |
|--------------------------|-------|-------|--------|-------|------------|-------------------|
| Hardware model | GF100 | GF104 | GK104 | GK110 | | |
| Compute Capability (CCC) | 2.0 | 2.1 | 3.0 | 3.5 | | |
| Dynamic Parallelism | No | No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

What is 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.



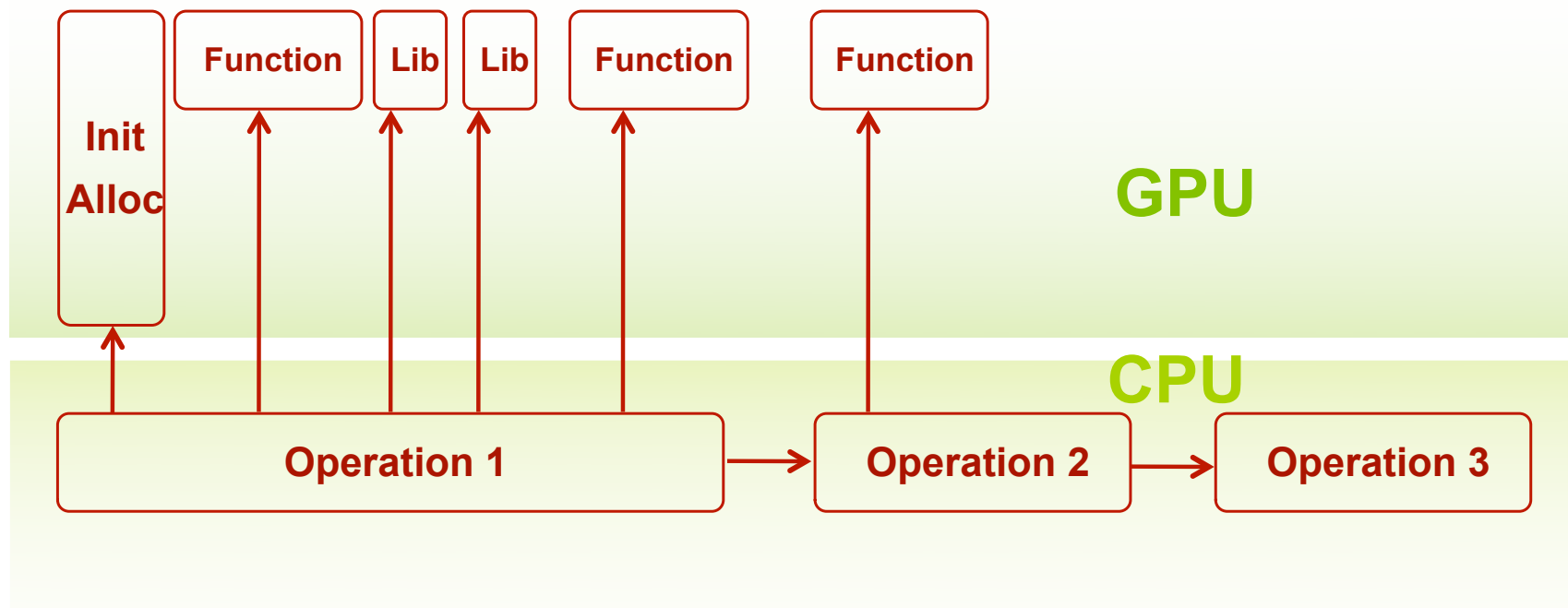
Fermi: Only CPU
can generate GPU work.



Kepler: GPU can
generate work for itself.

The way we did things in the pre-Kepler era: The GPU is a slave for the CPU

- High data bandwidth for communications:
 - External: More than 10 GB/s (PCI-express 3).
 - Internal: More than 100 GB/s (GDDR5 video memory and 384 bits, which is like a six channel CPU architecture).

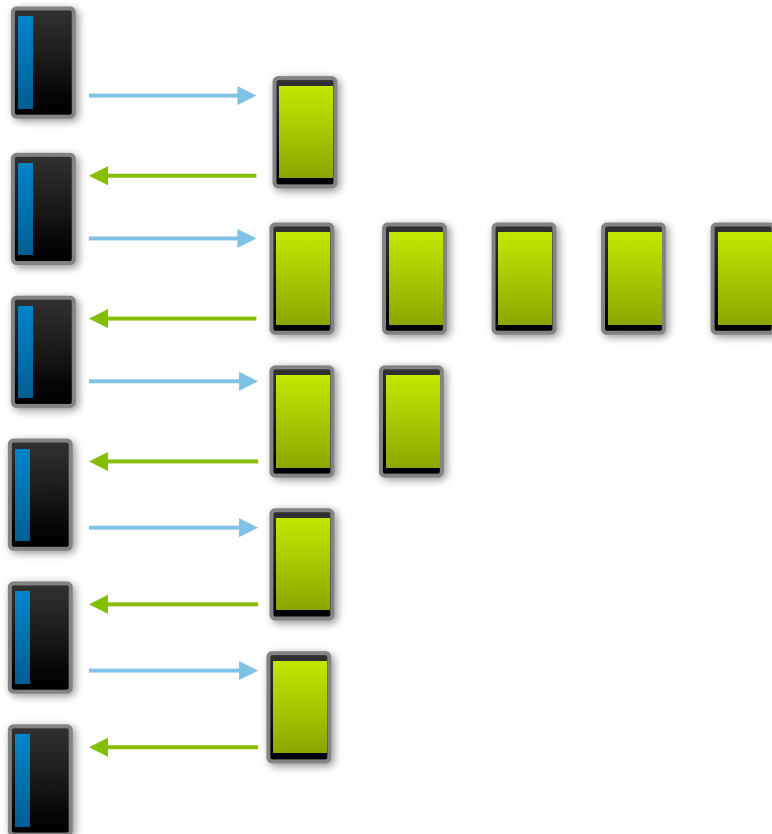


The way we do things in Kepler: GPUs launch their own kernels

The pre-Kepler GPU is a co-processor

CPU

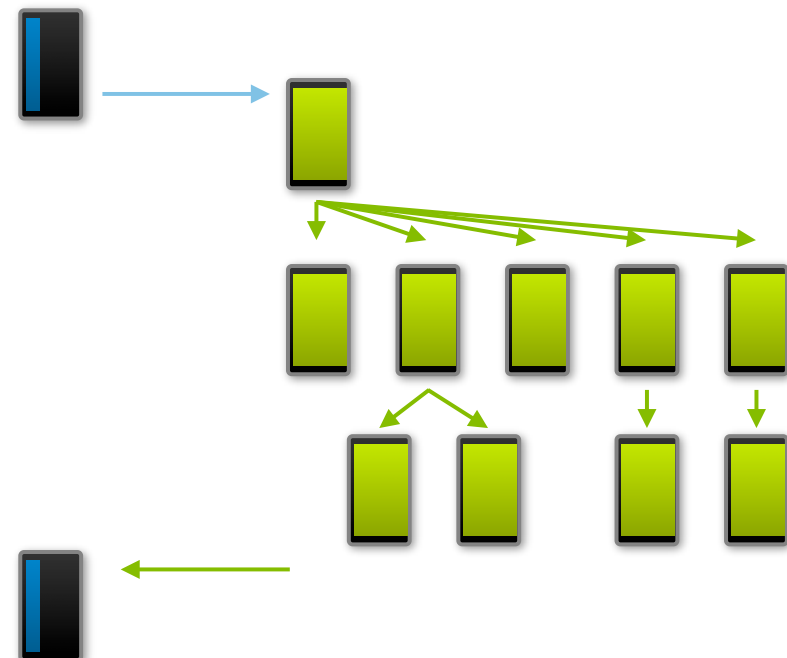
GPU



The Kepler GPU is autonomous:
Dynamic parallelism

CPU

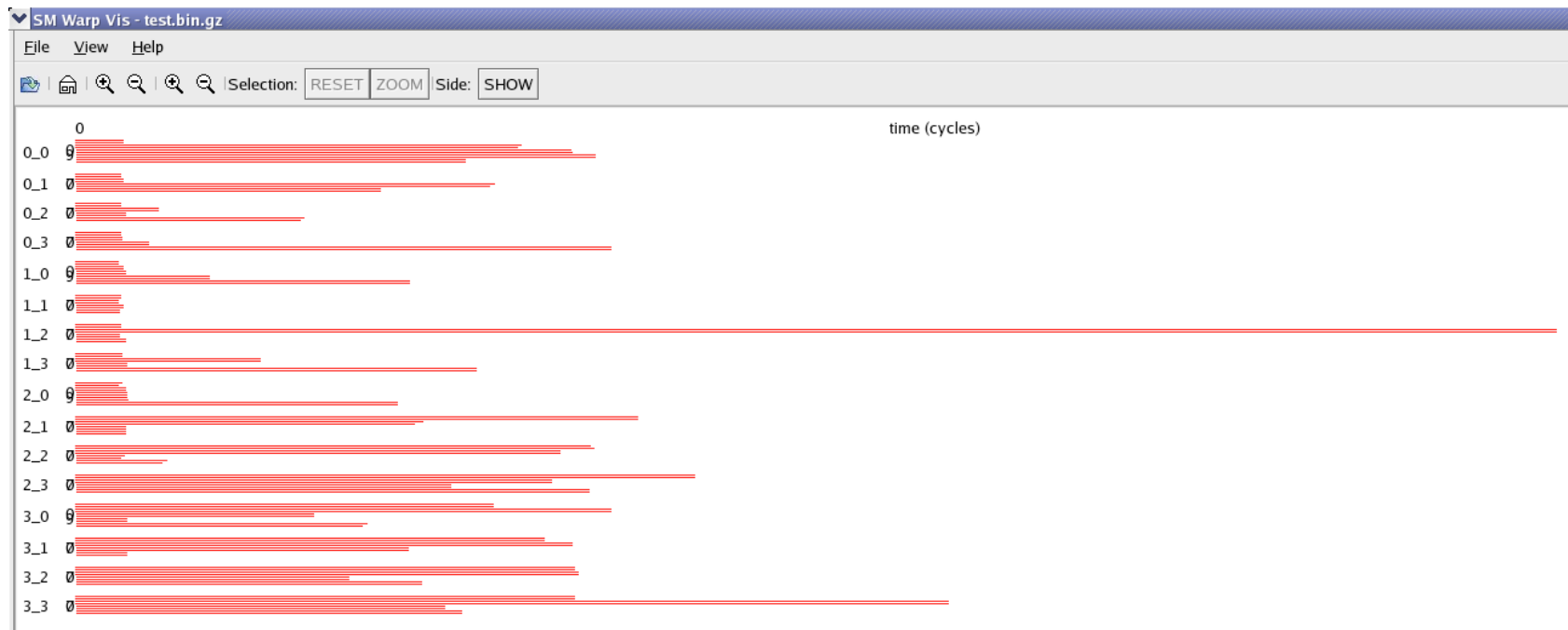
GPU



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

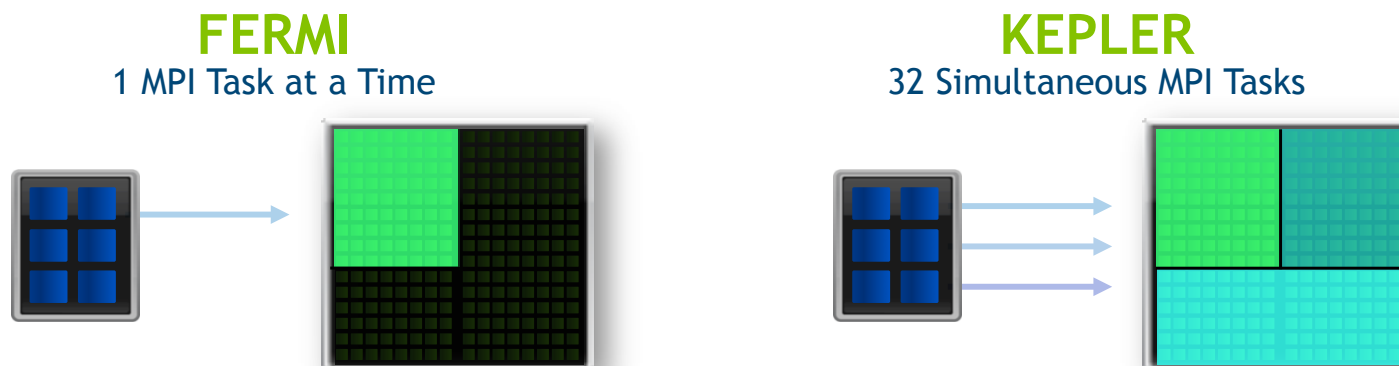
Watching the warps behaviour, we realize the GPU is far from being a regular processor

- Plenty of factors, unpredictable at run time, may transform the workload balance among multiprocessors into an impossible goal.
- Look at the duration of 8 warps on each SM for the G80:

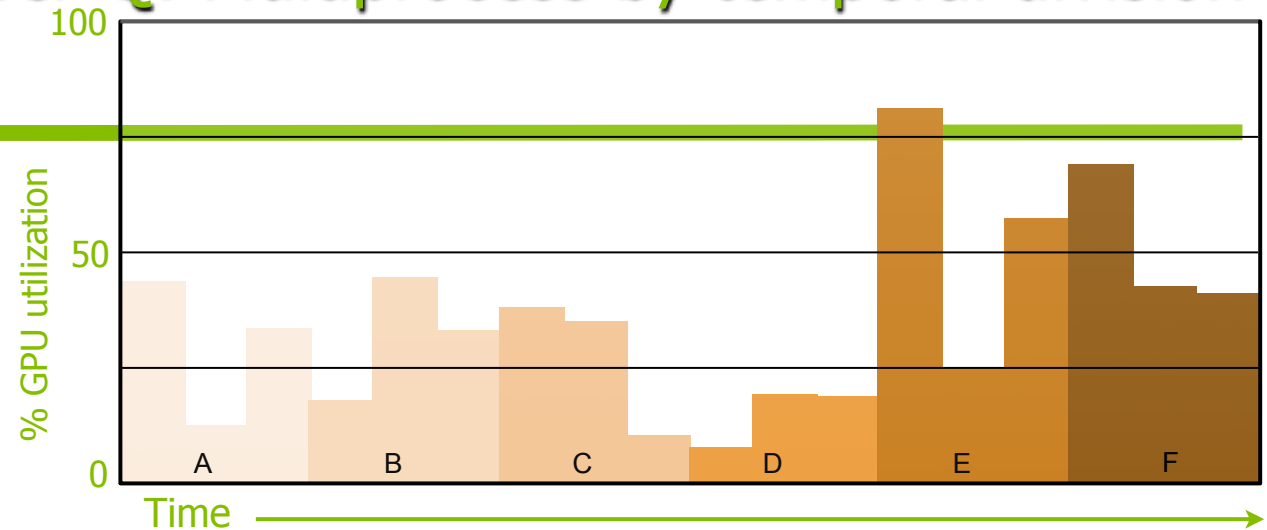
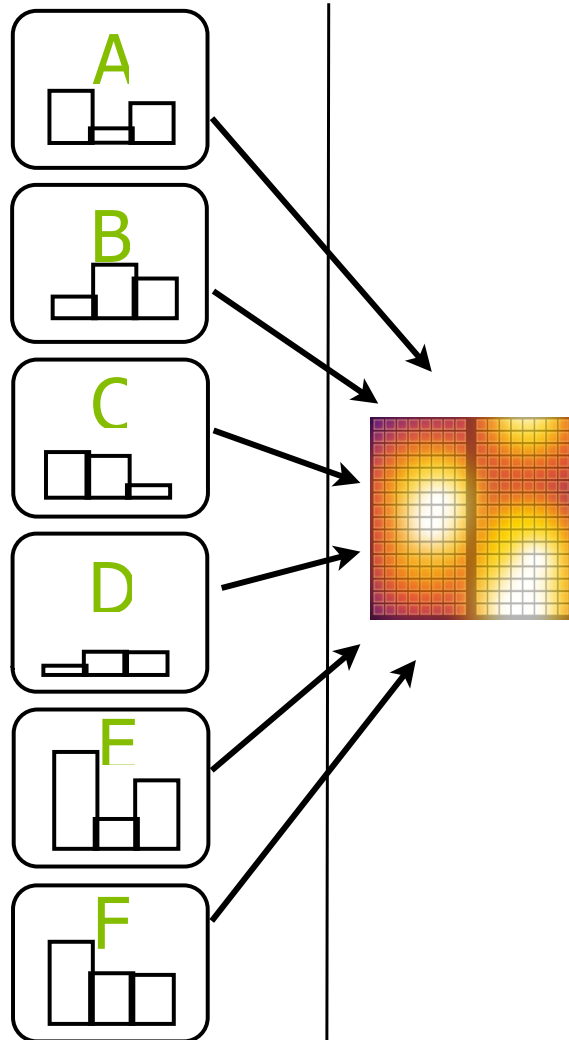


Hyper-Q

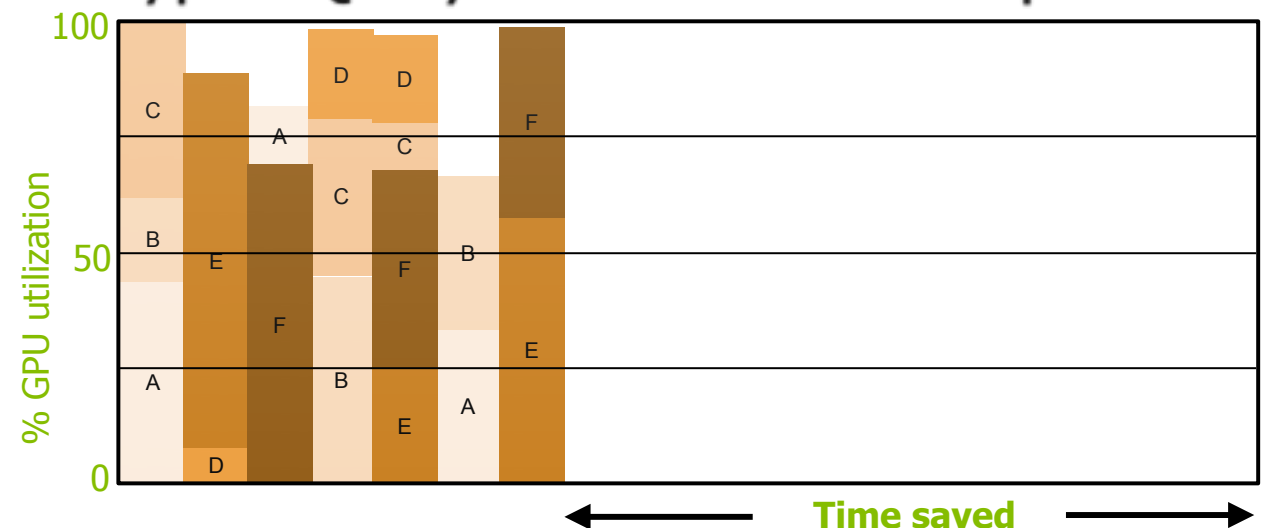
- In Fermi, several CPU processes can send thread blocks to the same GPU, but a kernel cannot start its execution until the previous one has finished.
- In Kepler, we can execute simultaneously up to 32 kernels launched from different:
 - MPI processes, CPU threads (POSIX threads) or CUDA streams.
-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



Without Hyper-Q: Multiprocess by temporal division



With Hyper-Q: Symultaneous multiprocess

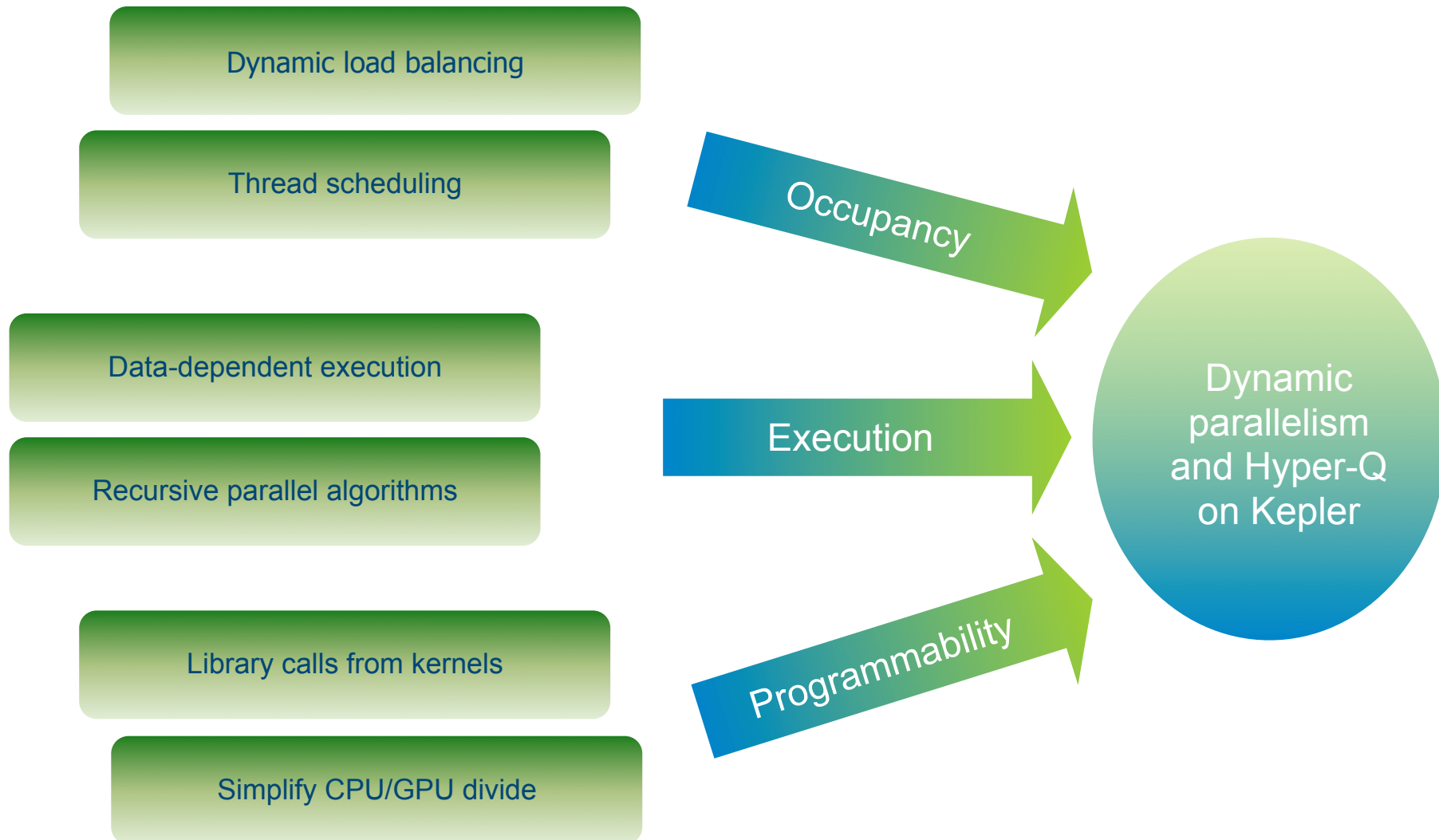


CPU processes... mapped on GPU

5. Exploiting on Kepler the new capabilities



Six ways to improve our codes on Kepler



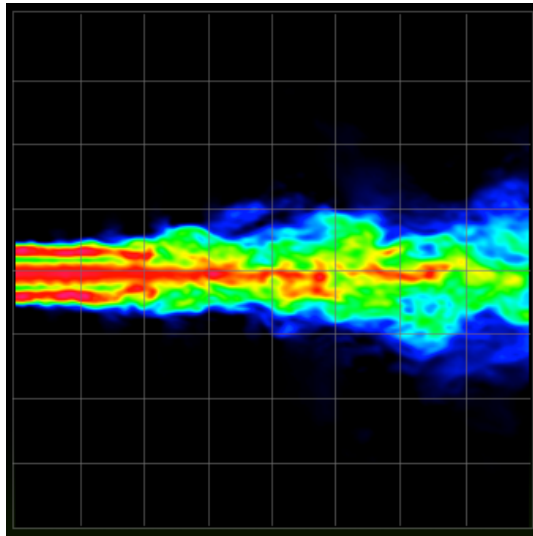
A large, stylized graphic of an eye, composed of concentric, curved lines in a light green color, centered on the slide. The right half of the slide has a solid green background, while the left half is white.

5.1. Dynamic load balancing

Dynamic work generation

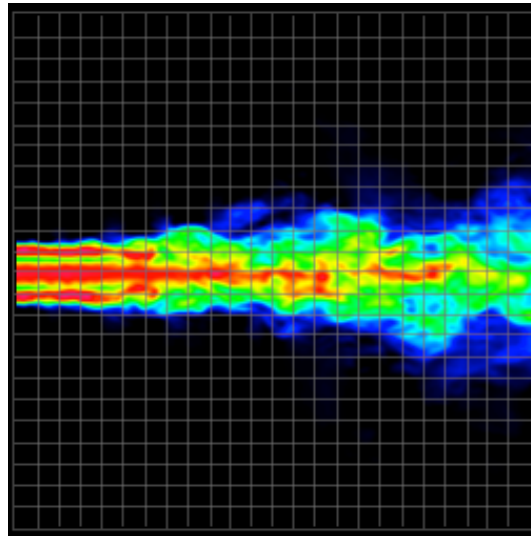
- Assign resources dynamically according to real-time demand, making easier the computation of irregular problems on GPU.
- It broadens the application scope where it can be useful.

Coarse grid



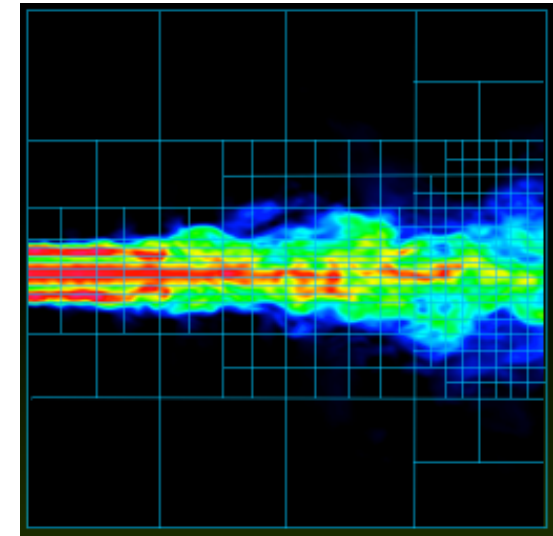
Higher performance,
lower accuracy

Fine grid



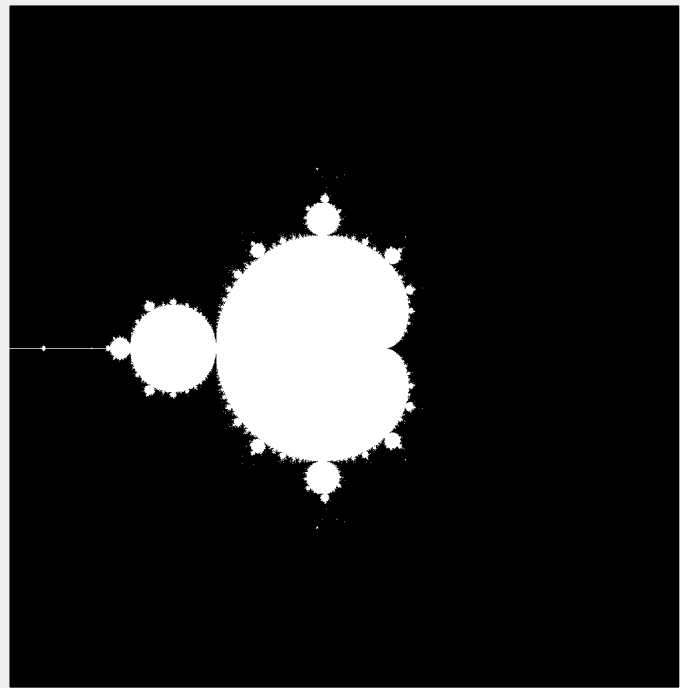
Lower performance,
higher accuracy

Dynamic grid



Target performance
where accuracy is required

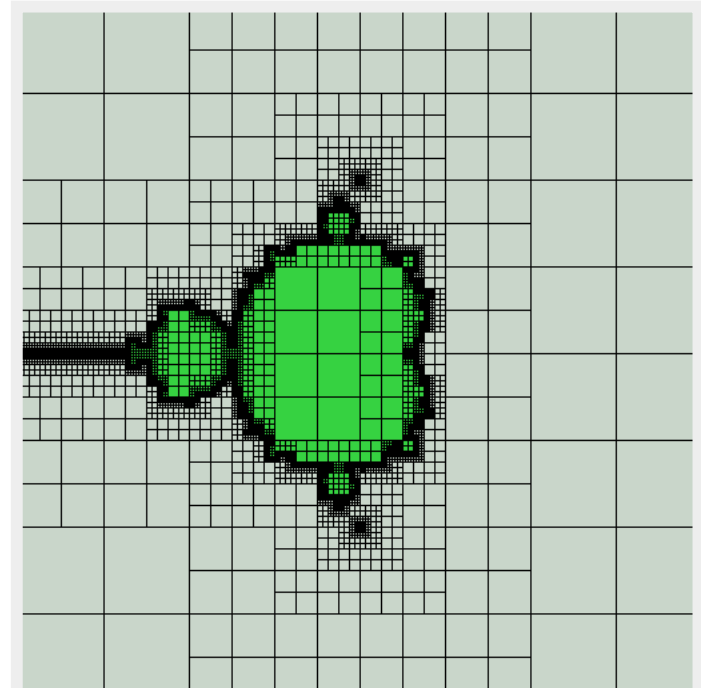
Deploy parallelism based on level of detail



CUDA until 2012:

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

Computational power
allocated to regions
of interest



CUDA on Kepler:

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



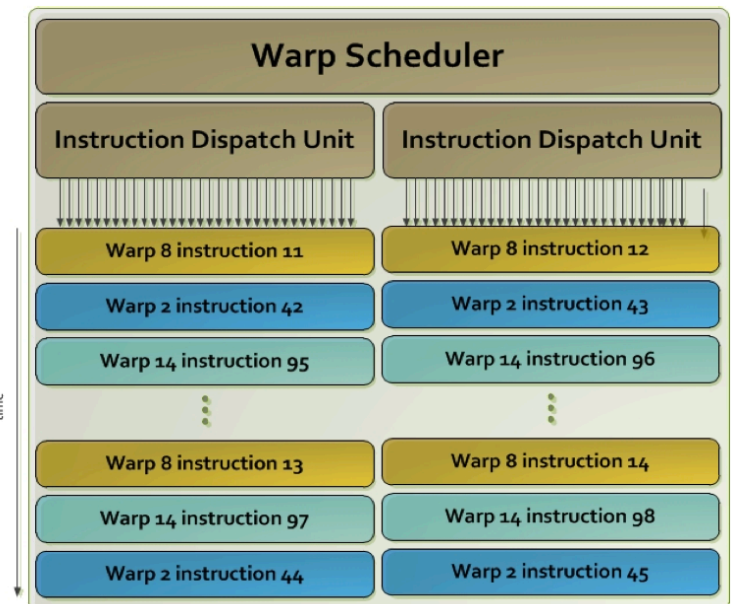
A large, stylized graphic of an eye, split vertically. The left half is white with a green outline, and the right half is solid green. The eye is looking towards the right.

5.2. Thread scheduling



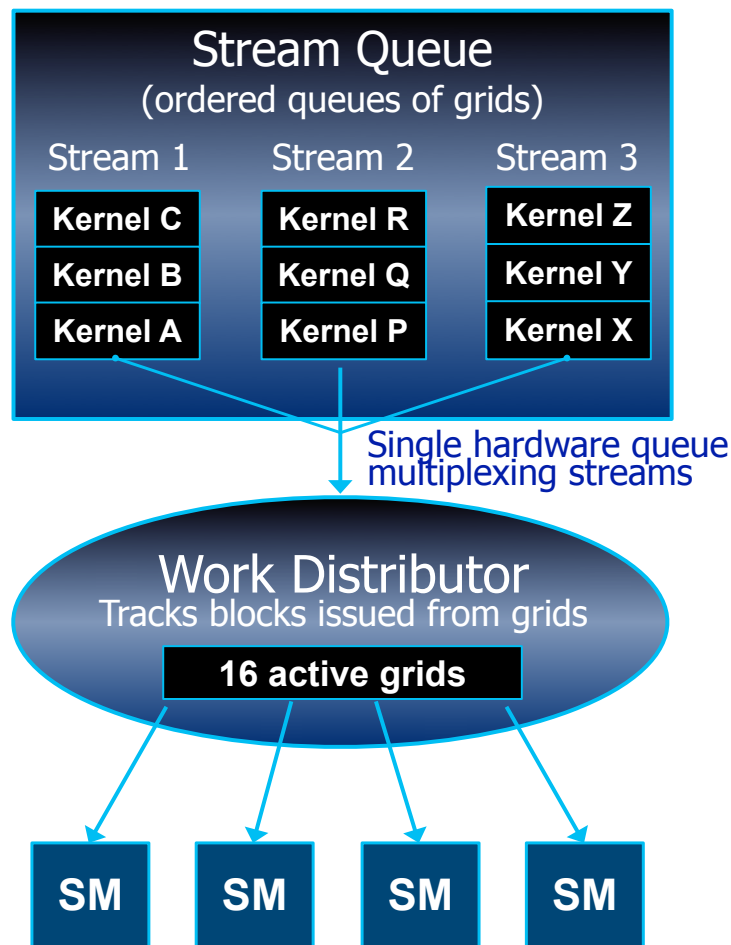
The way GigaThread scheduling works

- Each grid provides a number of blocks, which are assigned to SMXs (up to 32 blocks per SMX in Kepler, 16 in Fermi).
- Blocks are split into warps (groups) of 32 threads.
- Warps are issued for each instruction in kernel threads (up to 64 active warps in Kepler, 48 in Fermi). Kepler's snapshot:

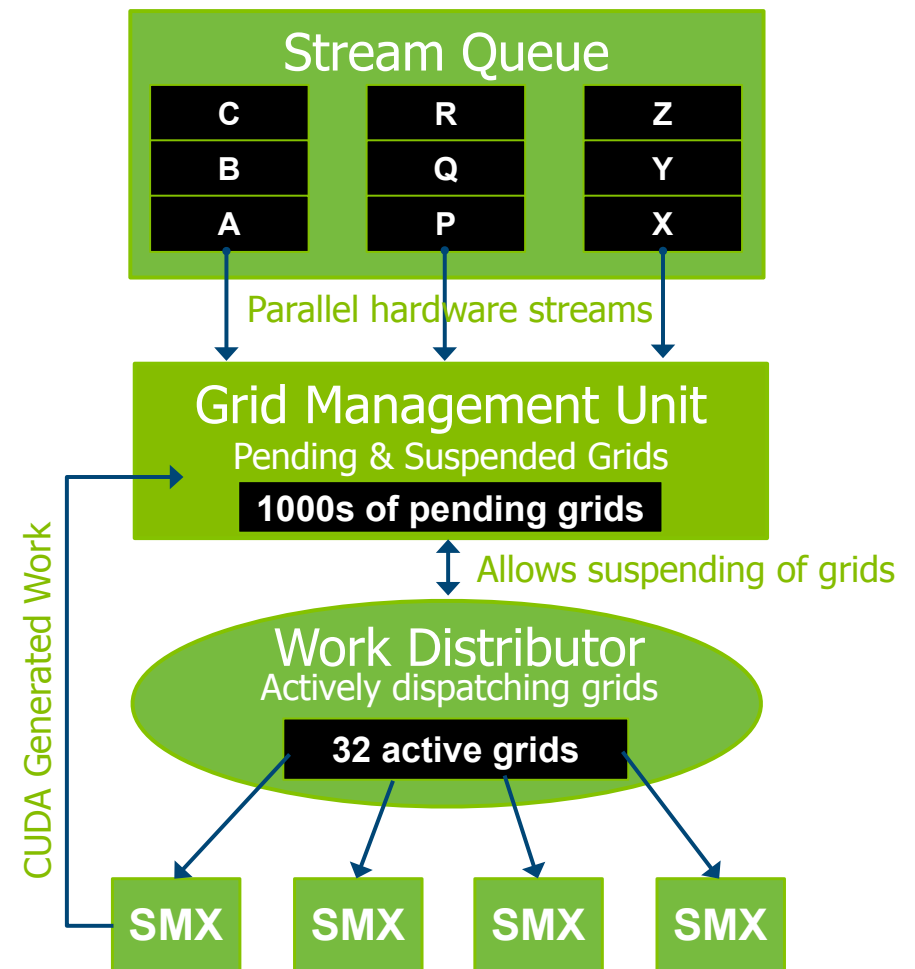


Grid management unit: Fermi vs. Kepler

Fermi



Kepler GK110

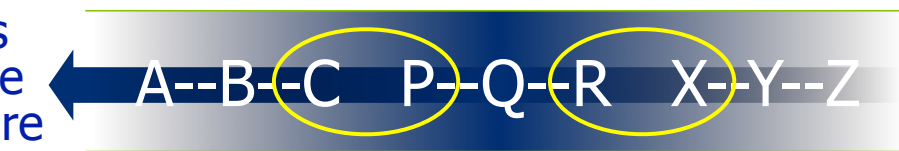


The relation between software and hardware queues

Fermi:

Up to 16 grids
can run at once
on GPU hardware

But CUDA streams multiplex into a single queue



Chances for overlapping: Only at stream edges

A -- B -- C

Stream 1

P -- Q -- R

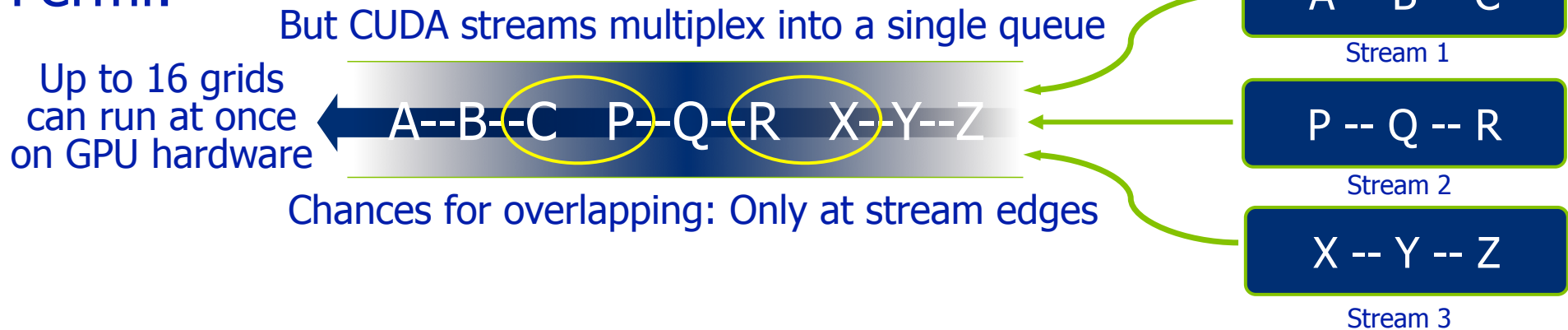
Stream 2

X -- Y -- Z

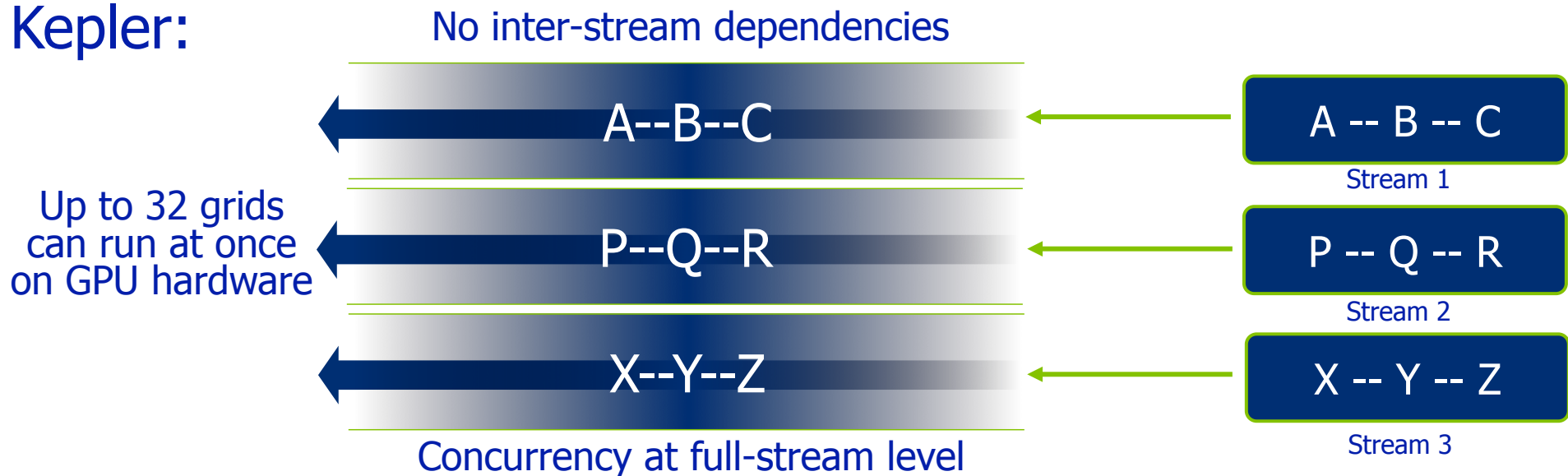
Stream 3

The relation between software and hardware queues

Fermi:



Kepler:



A case study for exploiting GPU concurrency in Fermi (15 SMs) and Kepler (15 SMXs)

- `mykernel <<< 100, 128, ... >>>` [We have a deficit in warps]
 - Launch 100 blocks of 128 threads (4 warps), that is, 400 warps.
 - There are 26.66 warps for each multiprocessor, either SM or SMX.
 - On Fermi: Up to 48 active warps (21 below the limit), which cannot be exploited.
 - On Kepler: Up to 64 active warps (37 below the limit), which can be activated from up to 32 kernel calls from MPI processes, POSIX threads or CUDA streams.
- `mykernel <<< 100, 384, ... >>>`
 - Launch 100 blocks of 384 threads (12 warps), that is, 1200 warps.
 - There are 80 warps for each multiprocessor. We've reached the max of 64 active warps, so 16 warps * 15 SMX = 240 warps wait on Kepler queues to be activated.
- `mykernel <<< 1000, 32, ... >>>` [We have a surplus in blocks]
 - 66.66 blocks for each SMX, but the max. is 16. <100, 320> better.

Lessons to learn (and trade-offs involved)

- Blocks big enough to avoid facing the limit of 16 per SMX.
 - But blocks consume shared memory, and allocating more shared memory means less blocks and more threads per block.
- Threads per block big enough to saturate the limit of 64 active warps per SMX.
 - But threads consume registers, and using many registers means less threads per block and more blocks.
- Hints:
 - Have at least 3-4 active blocks, each with at least 128 threads.
 - Smaller number of blocks when shared memory is critical, but...
 - ... abusing of shared memory hurts concurrency and latency hiding.

A comparison between instructions issue and execution (front-end vs. back-end)

| | 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.</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). A total of up to 4 concurrent warps.</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> |

● In Kepler, each SMX can issue 8 warp-instructions per cycle, but due to resources and dependencies limitations:

- 7 is the sustainable peak.
- 4-5 is a good amount for instruction-limited codes.
- Memory- or latency-bound codes by definition will reduce IPC (instrs. per cycle).

Great advantages of the GPU (vs. CPU) related to the CUDA work distributor

- Context switch is free because registers and shared memory are allocated exclusively to threads and blocks.
- The processor keeps busy as long as there are always many active warps to hide memory and dependencies stalls.
- Bottleneck is on the front-end, so schedulers are critical.

A large, stylized graphic of an eye, composed of concentric, curved lines in a light green color, centered on the slide. The right half of the slide has a solid green background, while the left half is white.

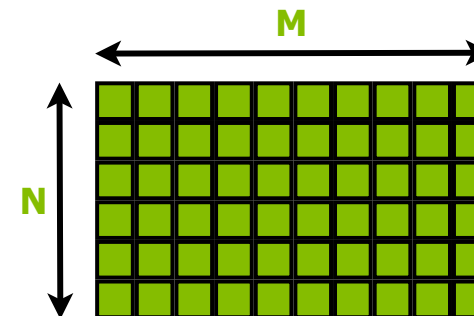
5.3. Data-dependent execution

Data-dependent parallelism

● The simplest possible parallel program:

- Loops are parallelizable.
- Workload is known at compile-time.

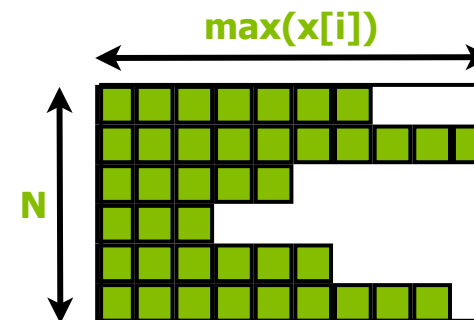
```
for i = 1 to N
  for j = 1 to M
    convolution (i, j);
```



● 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);
```



Poor solution #1: Oversubscription.

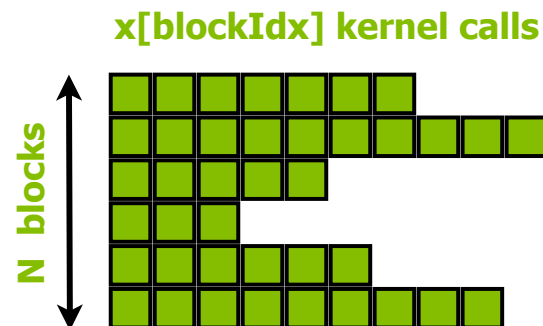
Poor solution #2: Serialization.

Now possible with dynamic parallelism: The two loops can be executed in parallel

● The CUDA program for Kepler:

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

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



Up to 24 nested loops
are allowed in CUDA 5.0.



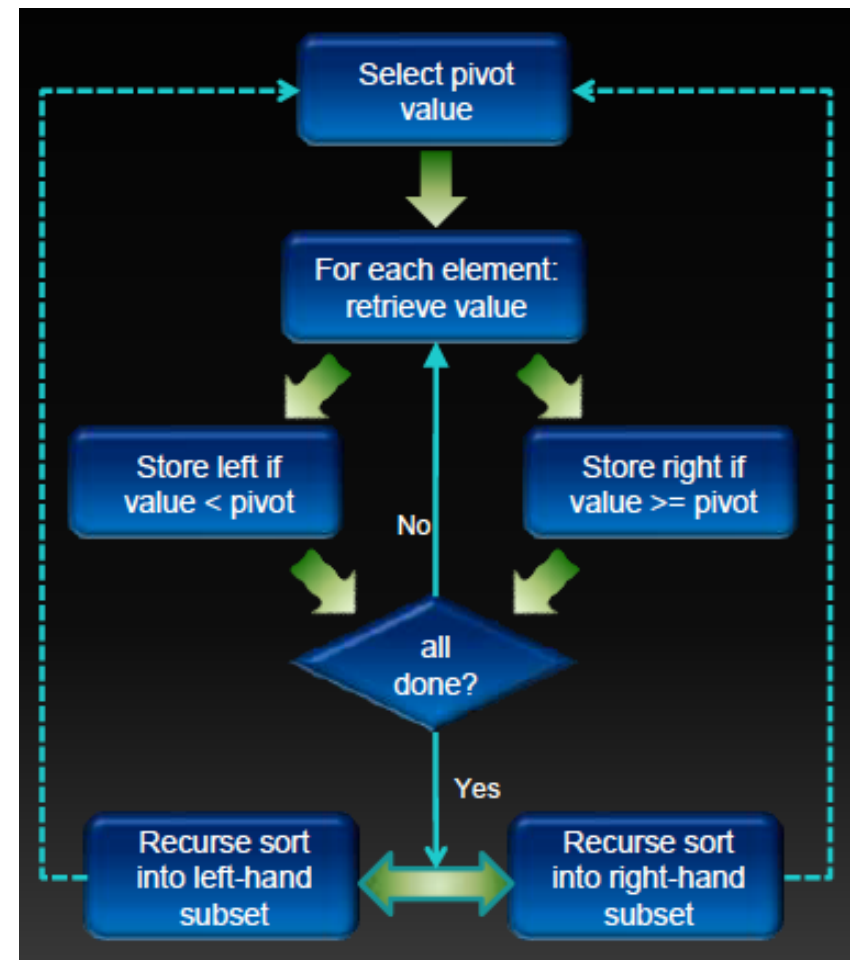
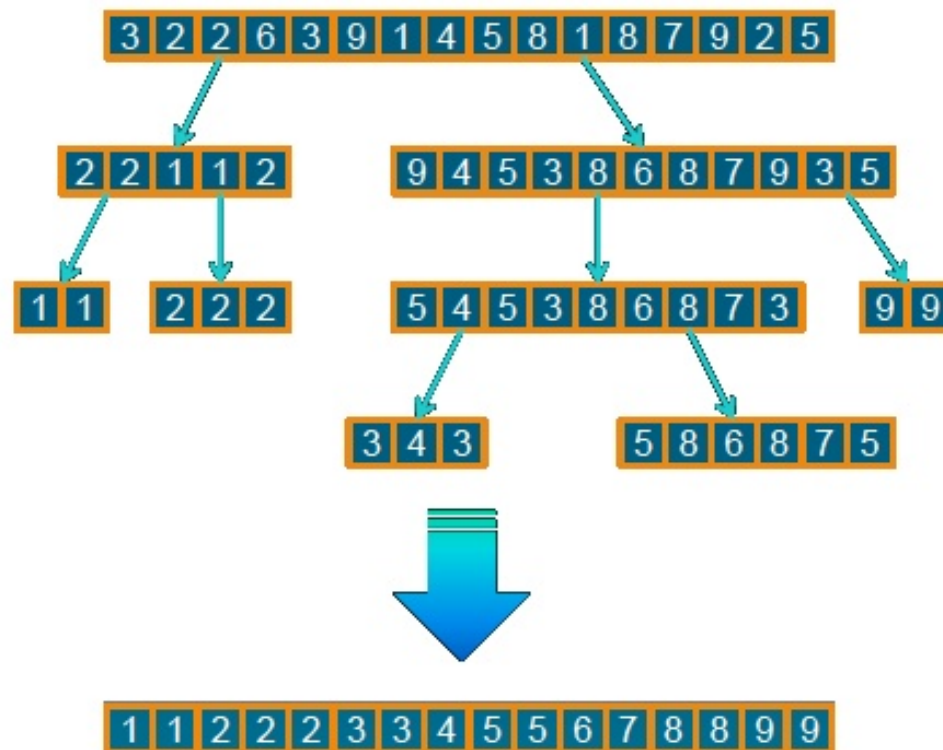
5.4. Recursive parallel algorithms

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.

A simple example of parallel recursion: Quicksort

- Typical divide-and-conquer algorithm hard to do on Fermi:
 - Entire data-dependent execution.
 - Recursively partition-and-sort data.



CUDA code for 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

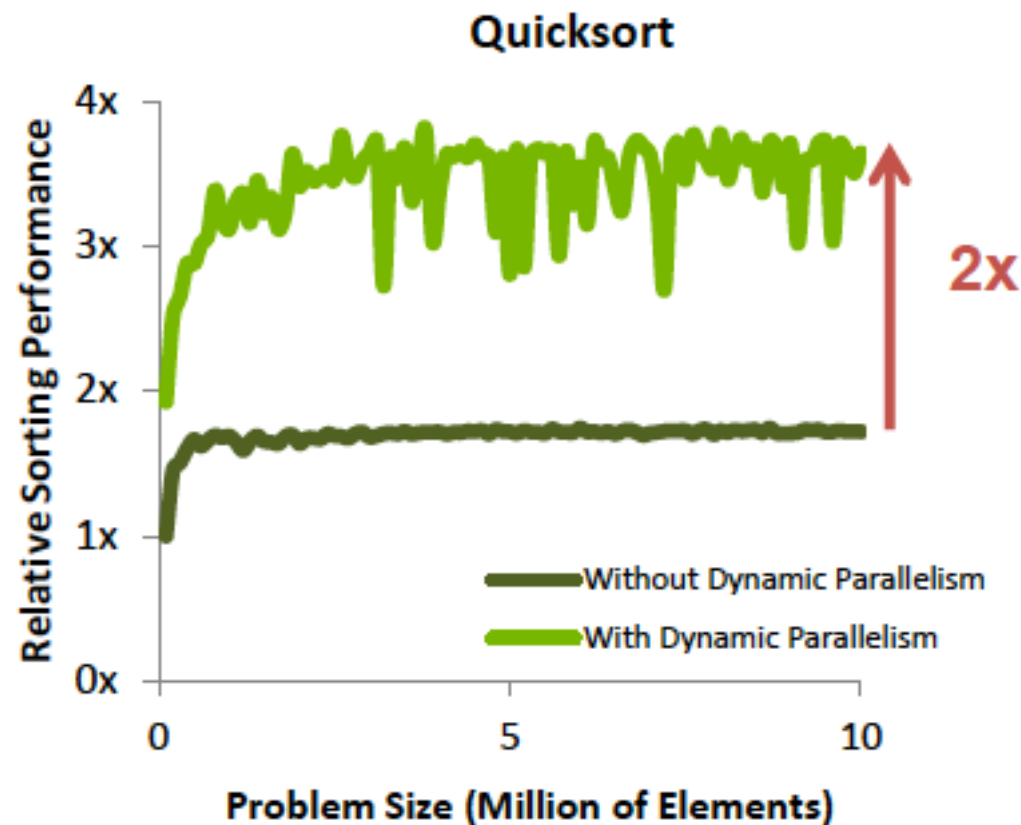
```
_global_ void qsort(int *data, int l, int r)
{
    int pivot = data[0];
    int *lptr = data+l, *rptr = data+r;
    // Partition data around pivot value
    partition(data, l, r, lptr, rptr, pivot);

    // 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

Experimental results for Quicksort

- The lines of code were reduced in half.
- Performance was improved by 2x.



A large, stylized eye logo in green and white, centered on the slide. The eye is composed of several concentric, curved lines that form the shape of an eye, with a smaller, similar eye shape nested inside the lower part of it.

5.5. Library calls from kernels

Programming model basics:

CUDA run-time syntax & 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

An example of simple library calls using cuBLAS (now available for CUDA 5.0)

```

__global__ void libraryCall(float *a,
                           float *b,
                           float *c)
{
    // All threads generate data
    createData(a, b);
    __syncthreads();

    // The first thread calls library
    if (threadIdx.x == 0) {
        cublasDgemm(a, b, c);
        cudaDeviceSynchronize();
    }

    // All threads wait for results
    __syncthreads();

    consumeData(c);
}
  
```

CPU launches kernel

Per-block data generation

Call of 3rd party library

3rd party library executes

Parallel use of result

The father-child relationship in CUDA blocks

```

__global__ void libraryCall(float *a,
                           float *b,
                           float *c)
{
    // All threads generate data
    createData(a, b);
    __syncthreads();

    // The first thread calls library
    if (threadIdx.x == 0) {
        cublasDgemm(a, b, c);
        cudaDeviceSynchronize();
    }

    // All threads wait for results
    __syncthreads();

    consumeData(c);
}
  
```


Per-thread execution

Single call to external library function:
 - The library will generate the child-block.
 - But we synchronize in the father-block.

Synchronize only launching threads:
 - Otherwise, race conditions may occur between father and child.

Father and child are different blocks, so:
 - Local and shared memory from father cannot be used in child.
 - Requires to copy values into global memory to be passed as kernel arguments to child.

All threads must wait before parallel data use

A large, stylized eye logo in green and white, centered on the slide. The eye is composed of several concentric, curved lines that form the shape of an eye, with a smaller, more detailed eye shape nested inside the lower part of the main eye.

5.6. Simplify the CPU/GPU division

A direct solver in matrix algebra: LU decomposition

Version for Fermi

| CPU side | | GPU side |
|---|---|-------------------------|
| <code>dgetrf(N, N) {</code> | | |
| <code>for j=1 to N {</code> | | |
| <code>for i=1 to 64 {</code> | | |
| <code>idamax<<<...>>></code> | → | <code>idamax();</code> |
| <code>memcpy</code> | ← | |
| <code>dswap<<<...>>></code> | → | <code>dswap();</code> |
| <code>memcpy</code> | ← | |
| <code>dscal<<<...>>></code> | → | <code>dscal();</code> |
| <code>dger<<<...>>></code> | → | <code>dger();</code> |
| <code>}</code> | | |
| <code>memcpy</code> | ← | |
| <code>dlaswap<<<...>>></code> | → | <code>dlaswap();</code> |
| <code>dtrsm<<<...>>></code> | → | <code>dtrsm();</code> |
| <code>dgemm<<<...>>></code> | → | <code>dgemm();</code> |
| <code>}</code> | | |
| <code>}</code> | | |

CPU fully occupied controlling launches

Version for Kepler

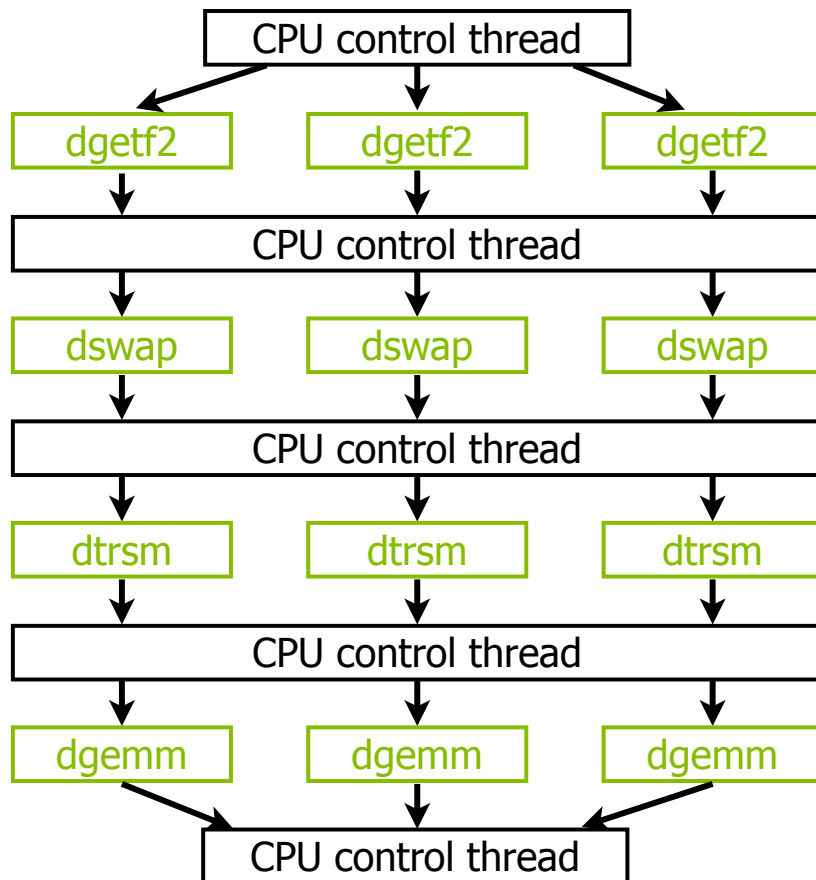
| CPU side | | GPU side |
|--|---|---|
| <code>dgetrf(N, N) {</code> | | |
| <code>dgetrf<<<...>>></code> | → | <code>dgetrf(N, N) {</code> |
| | | <code>for j=1 to N {</code> |
| | | <code>for i=1 to 64 {</code> |
| | | <code>idamax<<<...>>></code> |
| | | <code>dswap<<<...>>></code> |
| | | <code>dscal<<<...>>></code> |
| | | <code>dger<<<...>>></code> |
| | | <code>}</code> |
| | | <code>dlaswap<<<...>>></code> |
| | | <code>dtrsm<<<...>>></code> |
| | | <code>dgemm<<<...>>></code> |
| | | <code>}</code> |
| | | <code>}</code> |
| <code>synchronize();</code> | ← | |
| <code>}</code> | | |

Batched LU, release CPU for other work

Extended gains when our task involves thousands of LUs on different matrices

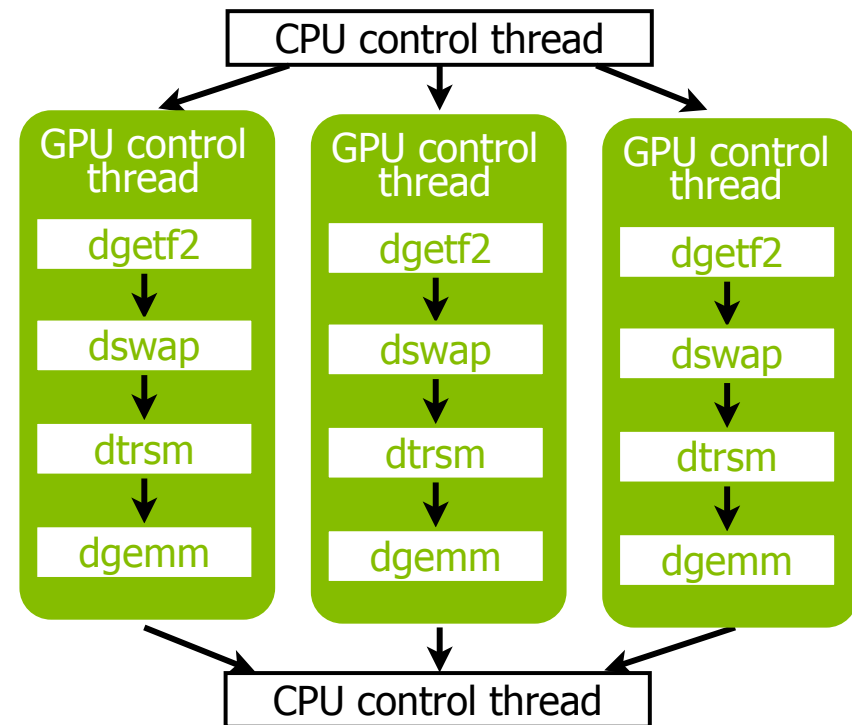
● CPU-controlled work batching:

- Serialize LU calls, or
- Face parallel P-threads limitations (10s).



● Batching via dynamic parallelism:

- Move top loops to GPU and launch 1000s of batches in parallel from GPU threads.



Concluding remarks

- Kepler represents the architectural design for 2012-2013, ready to host thousands of cores on a single die.
- It relies less on frequency and manufacturing process, more on power consumption and programmability, improving CUDA for irregular and dynamic applications.
- The GPU is more autonomous, but at the same time allows more interaction with the CPU.
- The memory hierarchy is also improved extensively, as well as the connection among GPUs.
- SMX-DRAM interconnect will play a decisive factor in future developments.

Bibliography

- Kepler whitepaper:

- <http://www.nvidia.com/object/nvidia-kepler.html>

- CUDA documentation:

- Best Practices Guide: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide>
 - Kepler Tuning Guide: <http://docs.nvidia.com/cuda/kepler-tuning-guide>

- Webinars (from GTC'12 to GTC'13, recent updates):

- <http://www.nvidia.com/object/webinar.html>

- Highly recommended:

- "CUDA 5 and beyond" [by Mark Harris].
 - "Compiling CUDA and other languages for GPUs" [Vinod Grover & Yuan Lin].
 - "New features in the CUDA programming model" [Stephen Jones & Lars Nyland].
 - "Introduction to dynamic parallelism" [Stephen Jones].
 - "Inside the Kepler Tesla K20 family" [Julia Levites & Stephen Jones].

Thanks for coming!

● You can always reach me in Spain at the Computer Architecture Department of the University of Malaga:

● e-mail: ujaldon@uma.es

● Phone: +34 952 13 28 24.

● Web page: <http://manuel.ujaldon.es>
(english/spanish versions available).

● Or, more specifically on GPUs, visit my web page as Nvidia CUDA Fellow:

● <http://research.nvidia.com/users/manuel-ujaldon>

