GPU Computing

Laurea **Magistrale in Informatica** - AA 2019/20

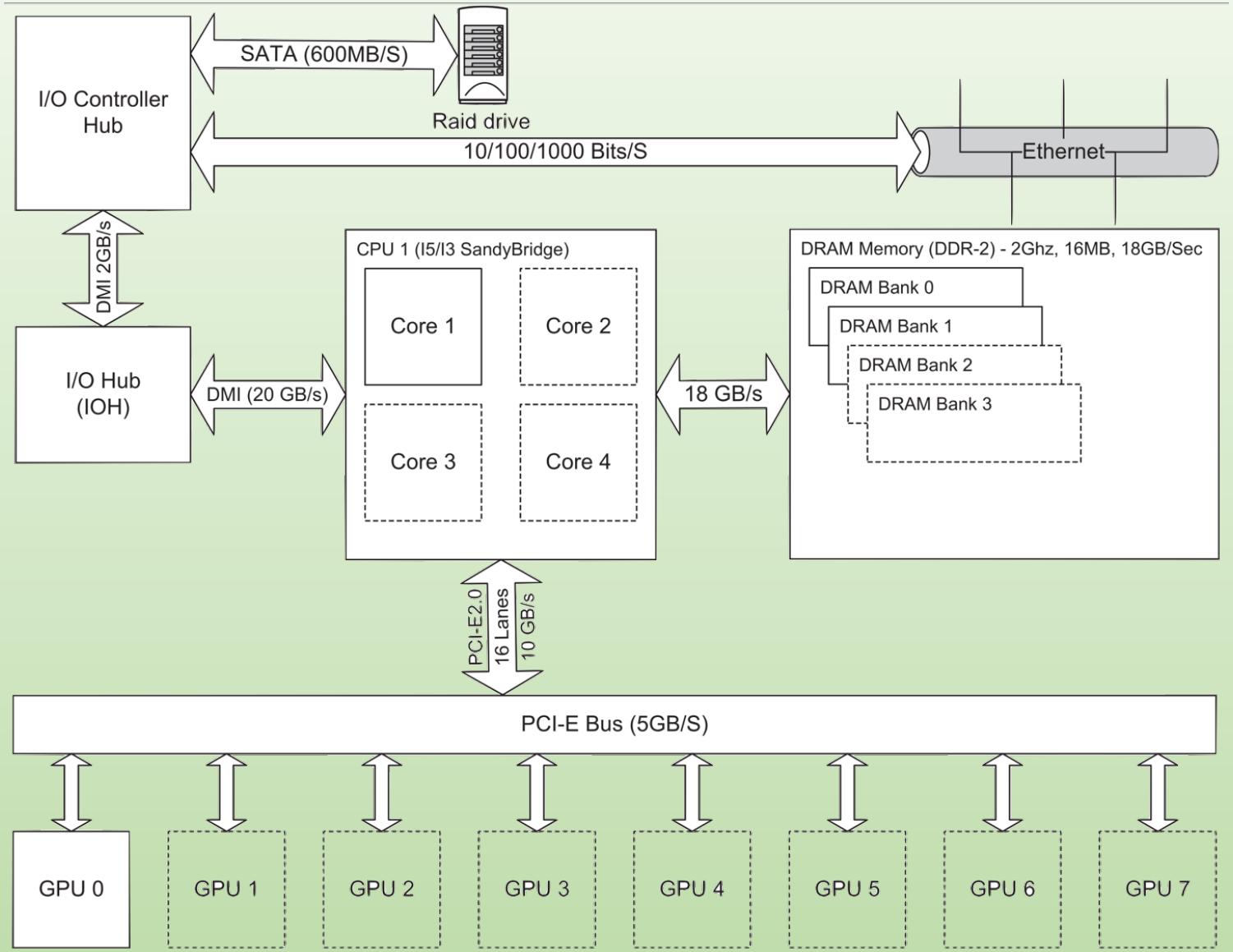
Docente **G. Grossi**

Lezione 4 – Architettura delle GPU

Sommario

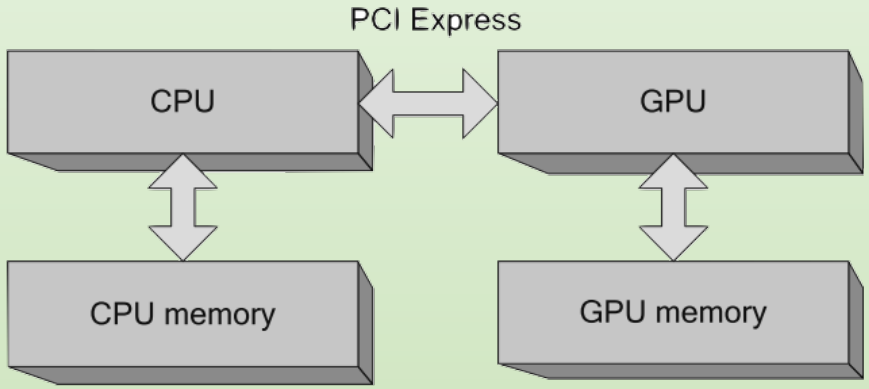
- ✓Elementi base dell’architettura GPU
- ✓Le famiglie di GPU e l’evoluzione delle (micro)architetture
- ✓Prestazioni legate alle risorse architetturali
- ✓Parallelismo dinamico e Hyper-Q
- ✓Parallelismo dinamico applicato a prodotti MQDB

Processori ed evoluzione HW (Intel I7 Nehalem)



Elementi architettura base CPU/GPU

- ✓ ES. ogni “lane” del **PCI Express 2.0** può teoricamente trasmettere fino a 500MB/s di bandwidth (il numero di line per una data periferica può essere 1, 4, 8, or 16)
- ✓ Le GPU richiedono la maggior bandwidth disponibile sulla piattaforma e sono progettate per essere collocate in slot di **16-lane PCIe** (500 x 16 = 8 GB)
- ✓ Mediante tecniche di packet overhead, i 8GB/s teorici di bandwidth arrivano fino a circa **6GB/s** in pratica



Debutto delle varie architetture GPU

HW Arch.	Anno	Modello consumer
Tesla	2006	GeForce 8800 GTX (G80)
Fermi	2010	GeForce GTX 480 (GF100)
Kepler	2012	GeForce GTX 680 (GK104)
Maxwell	2014	GeForce GTX 980 (GM204)
Pascal	2016	GeForce GTX 1080 (GP104)
Volta	2017	GeForce GTX 1180 (GV104)
Turing	2018	GeForce RTX 20 series

Compute Capability

- ✓ Termine usato per descrivere la **versione hardware** degli acceleratori GPU che appartengono alle famiglie di dispositivi e che hanno una data architettura interna
- ✓ Nelle versioni successive l'HW aumenta numero e prestazioni di elementi quali core, memoria, cache, bus, etc, quindi il throughput complessivo
- ✓ I device (GPU) con lo stesso **major revision number** hanno la stessa architettura base:
  - **Tesla** class architecture is major version number 1
  - **Fermi** class architecture is major version number 2
  - **Kepler** class architecture is major version number 3
  - **Maxwell** class architecture is major version number 5
  - **Pascal** class architecture is major version number 6
  - **Volta**, **Turing** class architecture is major version number 7

Architetture, modelli e carat.

Architecture Name	Compute Capability	GPUs	Example Features
Tesla	1.0	GeForce 8800, Quadro FX 5600, Tesla C870	Base Functionality
	1.1	GeForce 9800, Quadro FX 4700 x2	Asynchronous Memory Transfers
	1.3	GeForce GTX 295, Quadro FX 5800, Tesla C1060	Double Precision
Fermi	2.0	GeForce GTX 480, Tesla C2050	R/W Memory Cache
Kepler	3.0	GeForce GTX 680, Tesla K10	Warp Shuffle Functions, PCI-e 3.0
	3.5	Tesla K20, K20X, and K40	Dynamic Parallelism
	3.7	Tesla K80	More registers and shared memory
Maxwell	5.0	GeForce GTX 750 Ti	New architecture
	5.2	GeForce GTX 970/980	More shared memory
Pascal	6.0	Tesla P100	Page Migration Engine
Volta	7.0	Tesla V100	Tensor
Turing	7.5	GeForce RTX 2080	Ray-tracing

https://developer.nvidia.com/cuda-gpus#compute

GPUs Tesla per HPC

Tesla Workstation Products

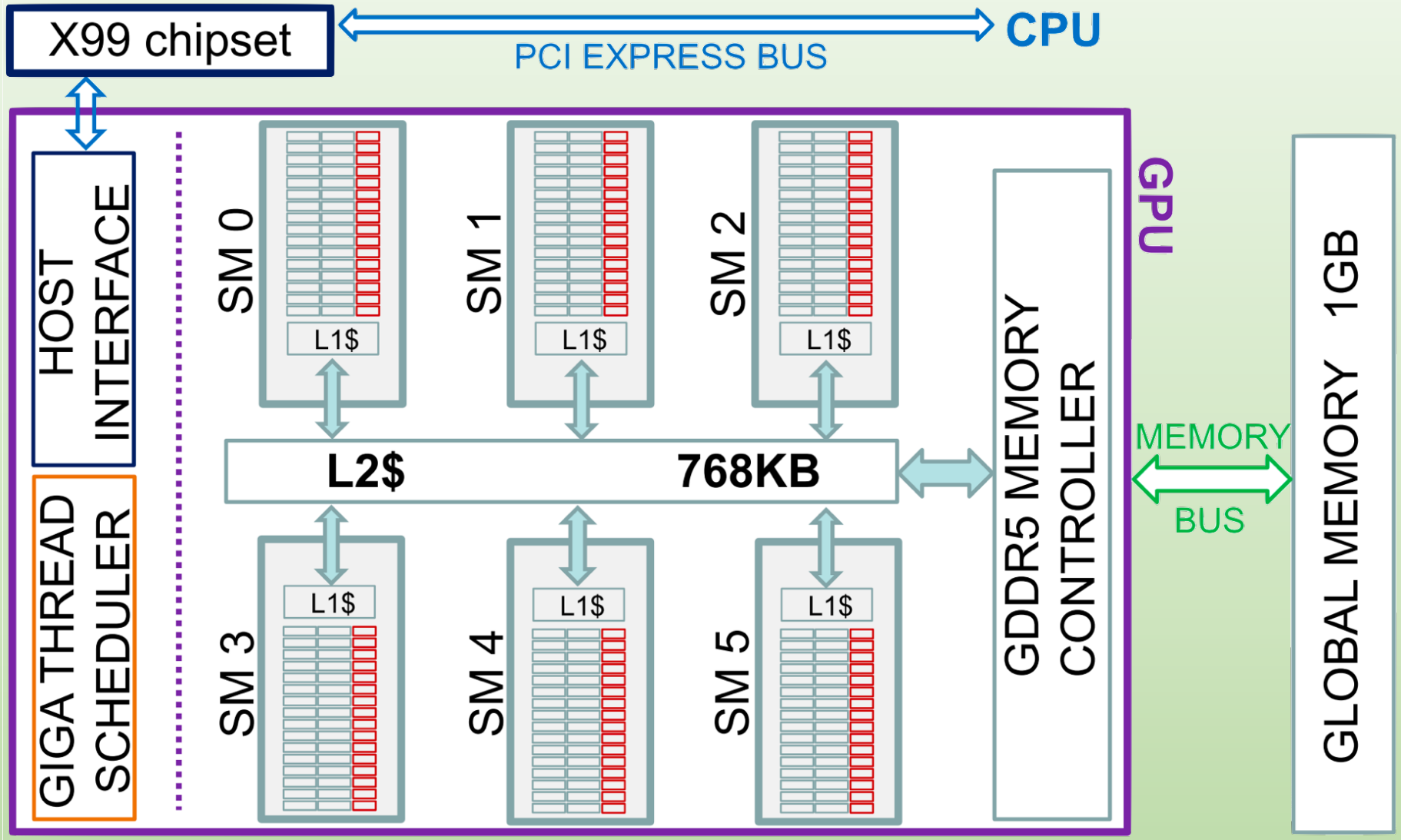
GPU	Compute Capability
Tesla K80	3.7
Tesla K40	3.5
Tesla K20	3.5
Tesla C2075	2.0
Tesla C2050/C2070	2.0

Tesla Data Center Products

GPU	Compute Capability
Tesla T4	7.5
Tesla V100	7.0
Tesla P100	6.0
Tesla P40	6.1
Tesla P4	6.1
Tesla M60	5.2
Tesla M40	5.2
Tesla K80	3.7
Tesla K40	3.5
Tesla K20	3.5
Tesla K10	3.0

https://developer.nvidia.com/cuda-gpus#compute

Architettura interna della GPU (Fermi - GTX550Ti)



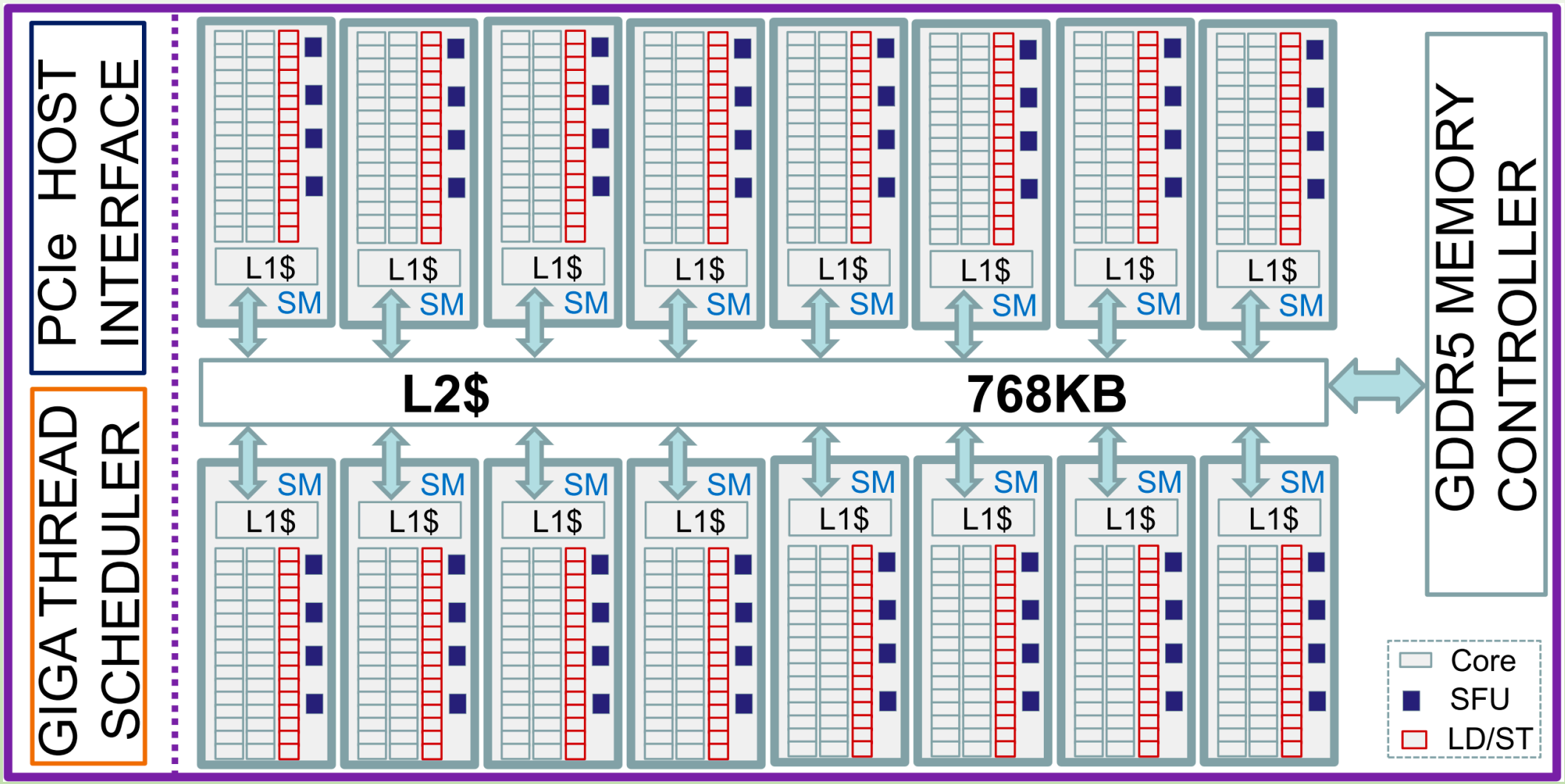
Host Interface

- ✓ **Controller** interno alla GPU **interfaccia** verso **bus PCIe**
- ✓ Consente alla GPU di **trasferire** dati dati **da** e **per** la **CPU** (~ I/O controller in CPU)
- ✓ Due tipi di memorie cache nella GPU: **L1$ interna** all'SM e **L2$ condivisa** tra SM
- ✓ Ogni cache **L1$** nell'SM è una **cache parallela** (serve 16 core alla volta)
- ✓ **L1$ c**ombina attività con le **unità Load/store** (con coda per accessi multipli)
- ✓ Un **memory controller** dedicato trasporta dati dentro e fuori dalla **GDDR5 global memory** and scaricandoli nella cache shared **L2$** (Last Level Cache - LLC)

Giga-Thread Scheduler

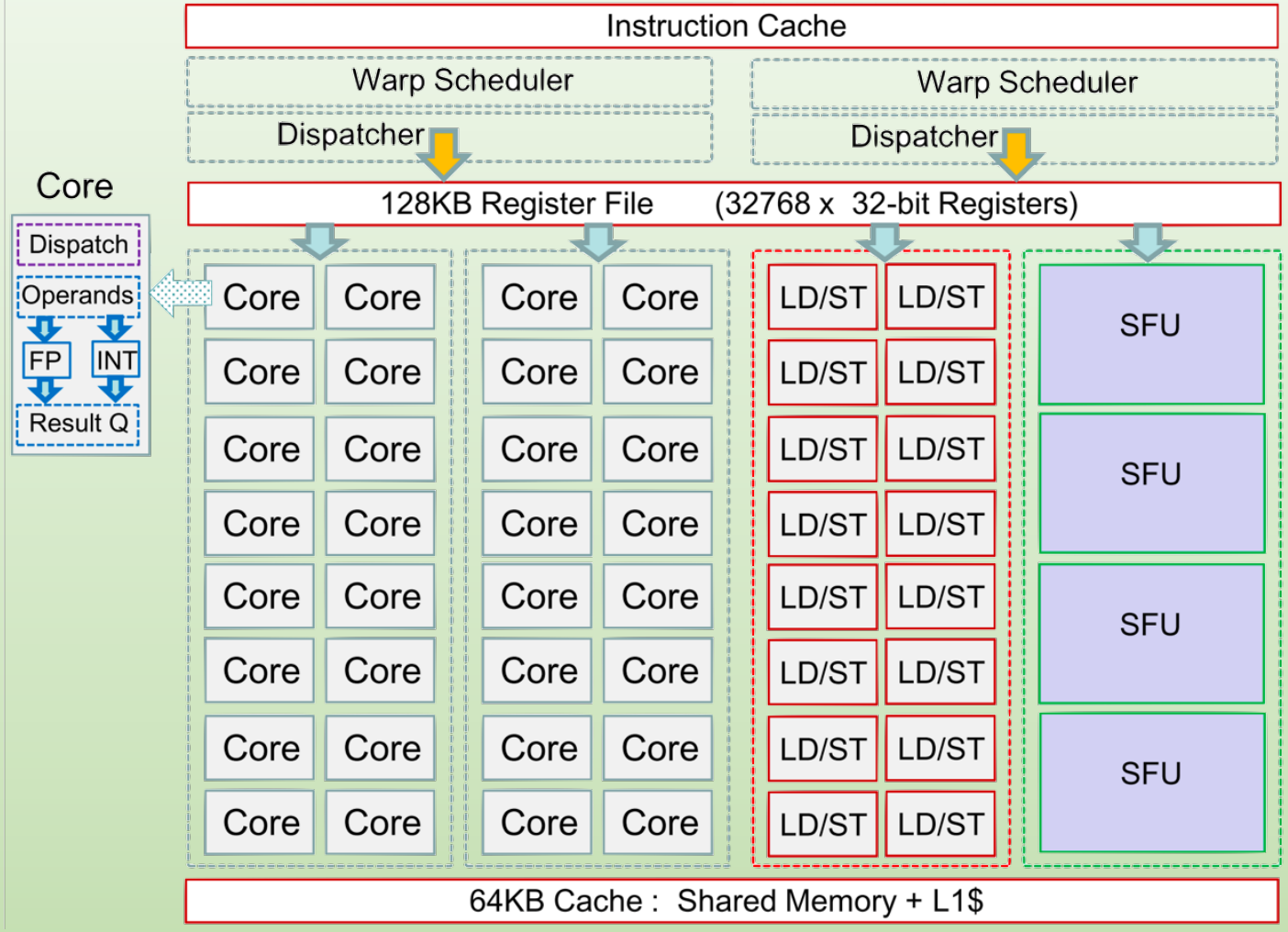
- ✓ Che cosa accade quando si Lancia un kernel con 2000 blocchi?
- ✓ Il **Giga-thread scheduler** si occupa di assegnarli ai vari SM disponibili (round robin policy):
  - es. **`Block0->SM0, Block1->SM1, ... Block1->SM5, Block6->SM0,…`**
- ✓ L'assegnamento di blocchi a SM è un'attività veloce… l'esecuzione dei blocchi molto più lenta
- ✓ Le vars **gridDim**, **blockIdx**, and **blockDim** sono passate dallo Giga Thread Scheduler ad ogni core della GPU che eseguirà il corrispondente blocco

Architettura FERMI (GF110)

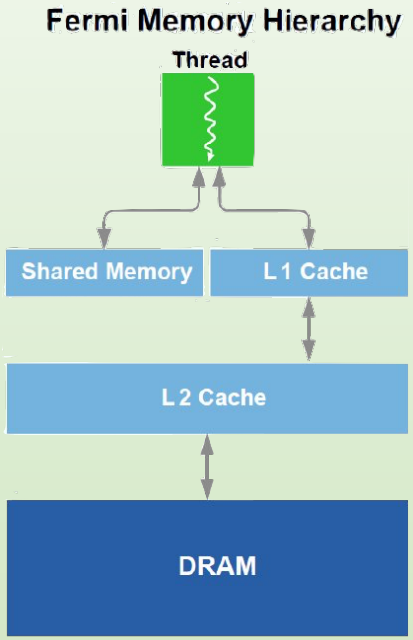


GF110 Fermi SM structure

- **32 cores** per SM
- **128KB register** file that contains **32,768 (32 K)** 32-bits
- This register file feeds operands to cores and **4 Special Function Units** (**SFU**).
- **16 Load/Store (LD/ST)** to queue memory load/store requests
- **64KB total cach**e memory is used for **L1$** and **shared memory**

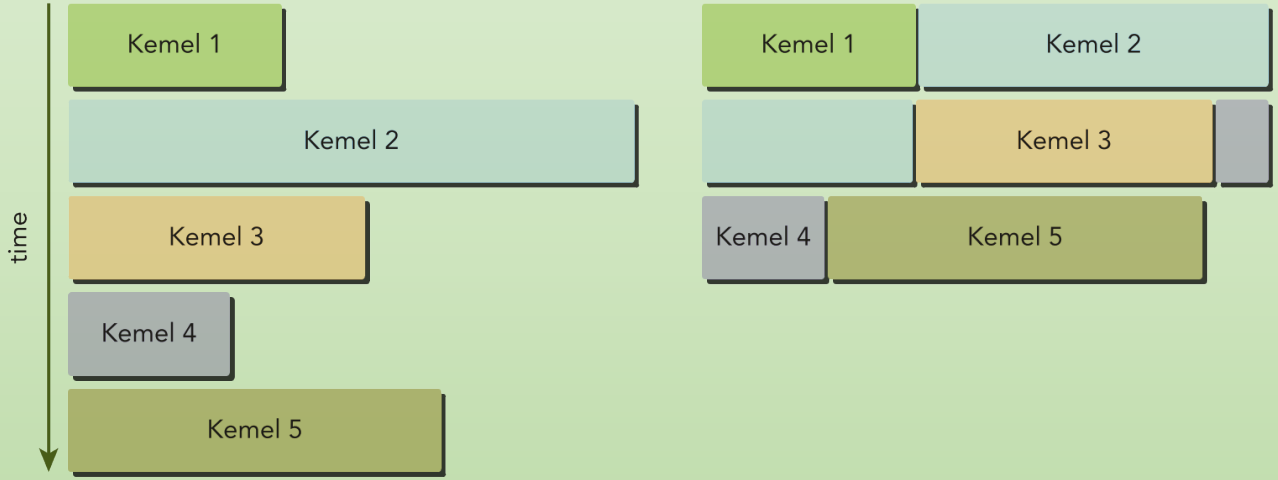


Fermi: kernel e memoria



- ✓ Una caratteristica chiave di Fermi è che la memoria di 64 KB on-chip è configurabile a runtime: può essere ripartita tra **shared memory** e **L1 cache**
- ✓ Per molte applicazioni HPC la shared memory consente grandi **prestazioni** perché consente ai thread di un blocco di cooperare, facilita il riuso di dati on-chip data e riduce il traffico off-chip
- ✓ CUDA fornisce **runtime API** che possono essere usate per **regolare** la quantità di **shared memory** e **L1 cache**

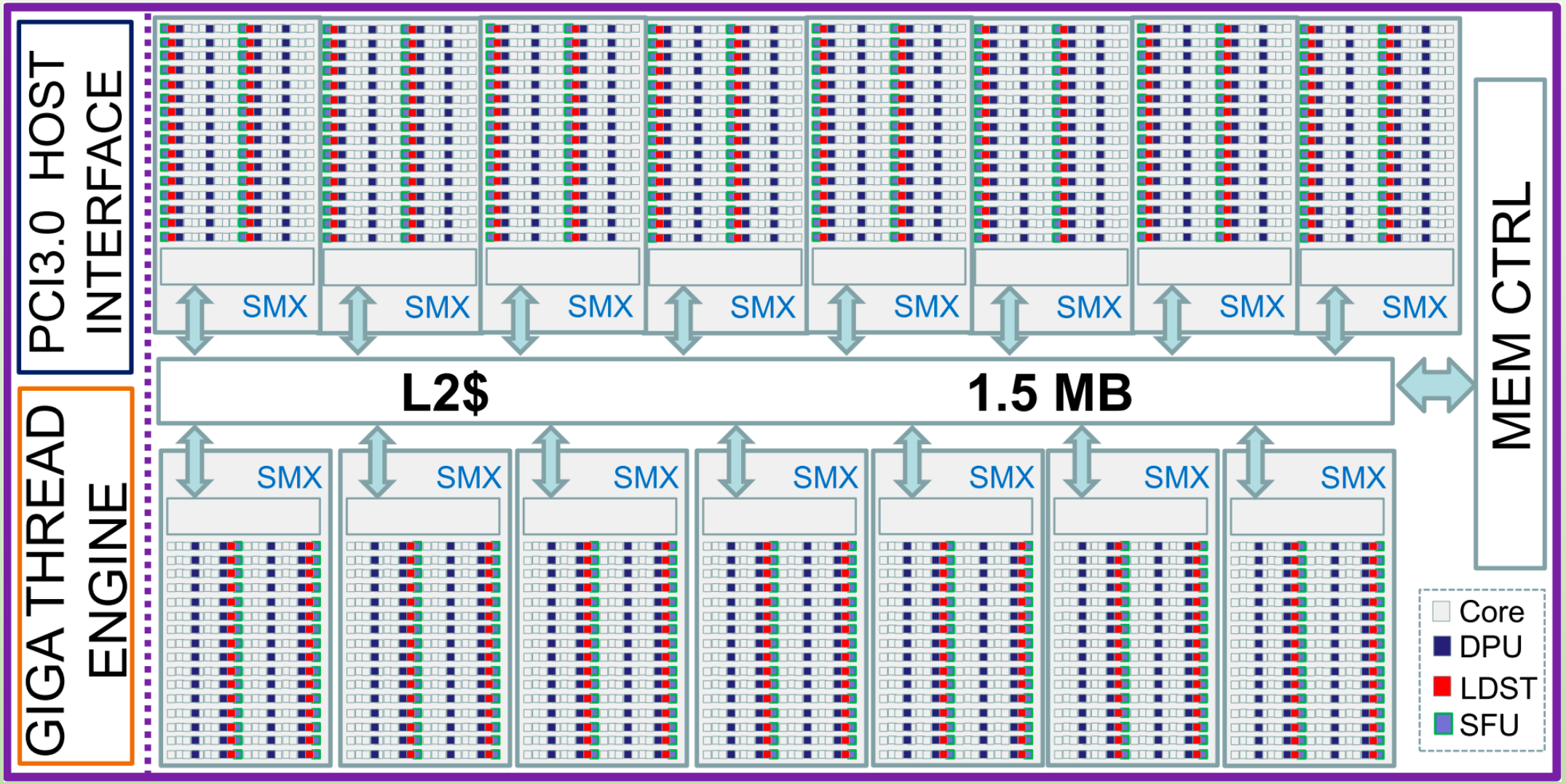
- ✓ Fermi supporta l'esecuzione **concorrente** di kernel: kernel multipli lanciati dalla stessa applicazione sulla stessa GPU
- ✓ Meglio in generale un numero alto di **piccoli kernel** per un utilizzo completo della GPU
- ✓ Fermi consente fino a **16 kernel** in esecuzione nello stesso tempo



Serial Kernel Execution

Concurrent Kernel Execution

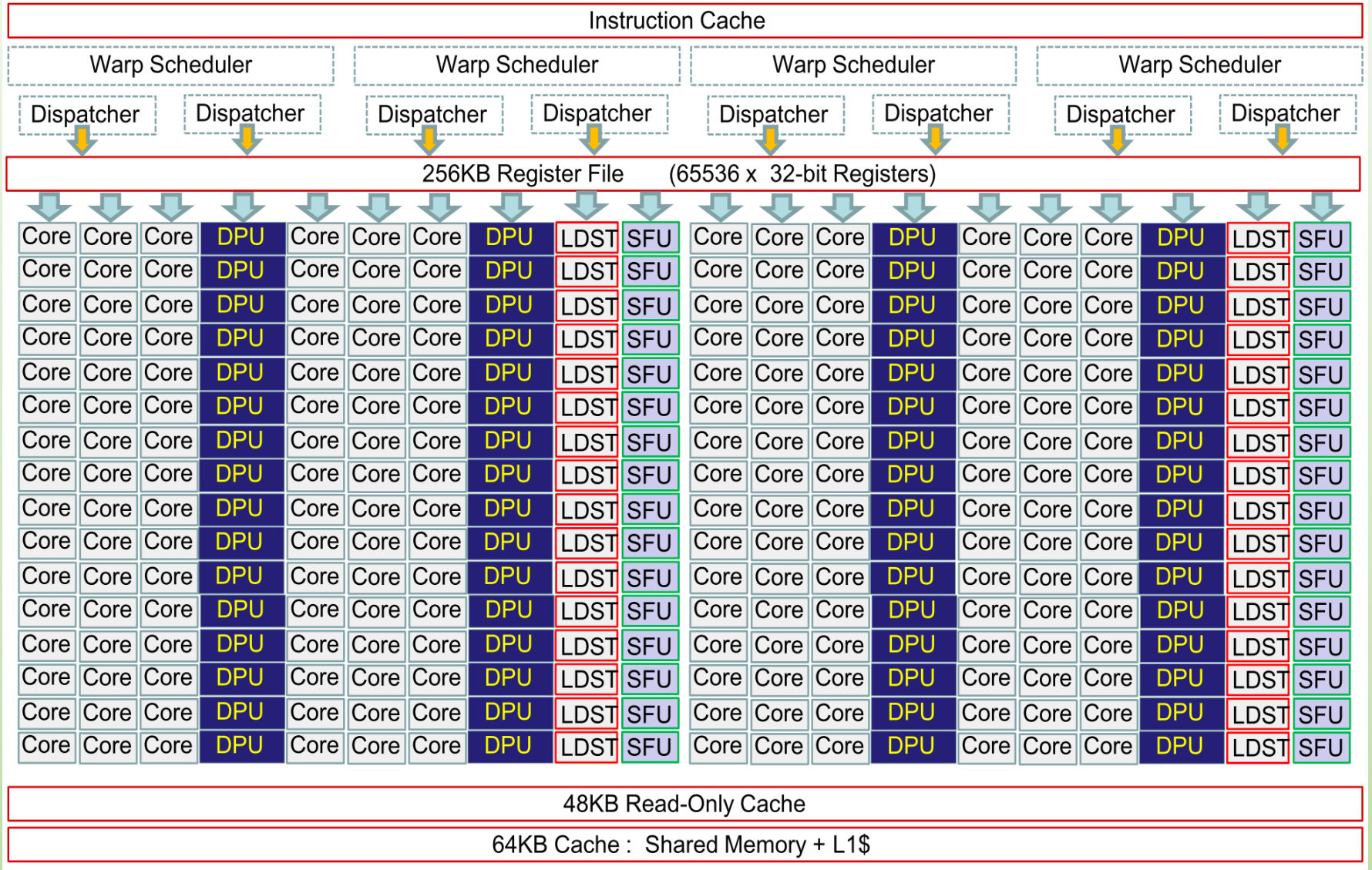
Architettura KEPLER



Kepler overview

- ✓ The **L2$** memory size is **doubled** (wrt Fermi) to 1536KB (1.5 MB)
- ✓ The host interface now supports **PCIe 3.0**
- ✓ Each SM is now called **SMX** and it has a completely **different** internal structure than **Fermi**
- ✓ The Giga Thread Scheduler is now called **Giga Thread Engine** (GTE)
- ✓ The SMX contains a different type of execution unit, named a **double precision unit** (**DPU**)
- ✓ **6 times** as many **cores** as **Fermi** (512 vs. 2880)… each **SMX** has **192** cores
- ✓ Having such heavily populated SMX units has interesting performance implications because the cores inside an SM (or SMX) share their **L1$**, **register file**, and another type of cache that is introduced, the **Read Only Cache**

GK110 Kepler SMX structure



Kepler SMX features

- **192 single-precision** CUDA cores
- **64 double-precision** units
- **32** special function units (**SFU**)
- **32** load/store units (**LD/ST**)
- **4** warp **schedulers**
- **8** instruction **dispatchers**
- **4 warps** executed **concurrently** on a single SMX
- **64** warps per SM **scheduled**
- **2048 threads** resident in a single SM at a time
- **64K register file** size
- more partitions of on-chip memory between **shared memory** and **L1$** cache
- 64KB cache is split between the two as either (16KB+48KB) or (48KB+16KB)
- more than **1 TFLOP** of peak **double-precision**
- **Dynamic Parallelism**
- **Hyper-Q**

FERMI vs KEPLER

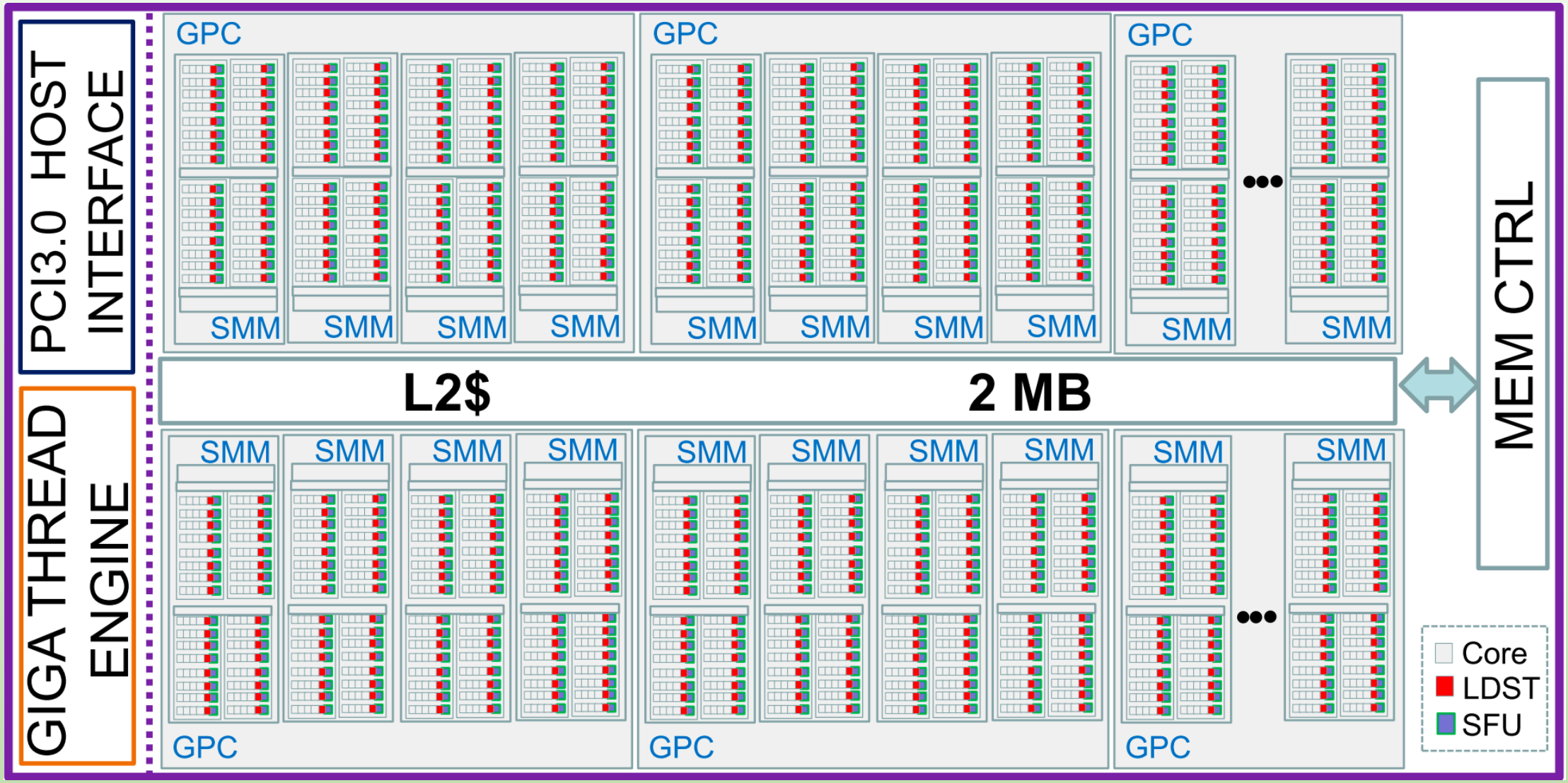
	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute capability 2. e 3.

ARCHITECTURE SPECIFICATIONS	COMPUTE CAPABILITY			
	2.0	2.1	3.0	3.5
Number of cores for integer and floating-point arithmetic function operations per multiprocessor	32	48	192	
Number of special function units for single-precision floating-point transcendental functions per multiprocessor	4	8	32	
Number of warp schedulers per multiprocessor	2		4	
Number of instructions issued at once by scheduler	1	2	2	
Number of load/store units per multiprocessor	16		32	

ARCHITECTURE SPECIFICATIONS	COMPUTE CAPABILITY			
	2.0	2.1	3.0	3.5
Load/Store address width	64 B		64 B	
L2 cache	768 K		1,536 K	
On-chip memory per multiprocessor	64 K		64 K	
Shared memory per multiprocessor (configurable)	48 K or 16 K		48 K/32 K/16 K	
L1 cache per multiprocessor (configurable)	16 K or 48 K		48 K/32 K/16 K	
Read-only data cache	N/A		48 K	
Global memory	Up to 6 GB		Up to 12 GB	

Architettura MAXWELL

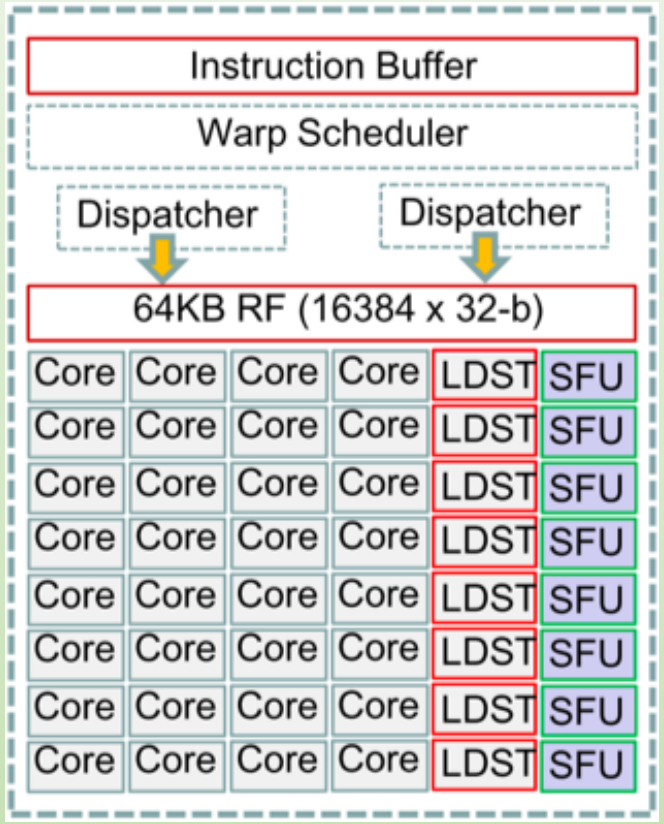


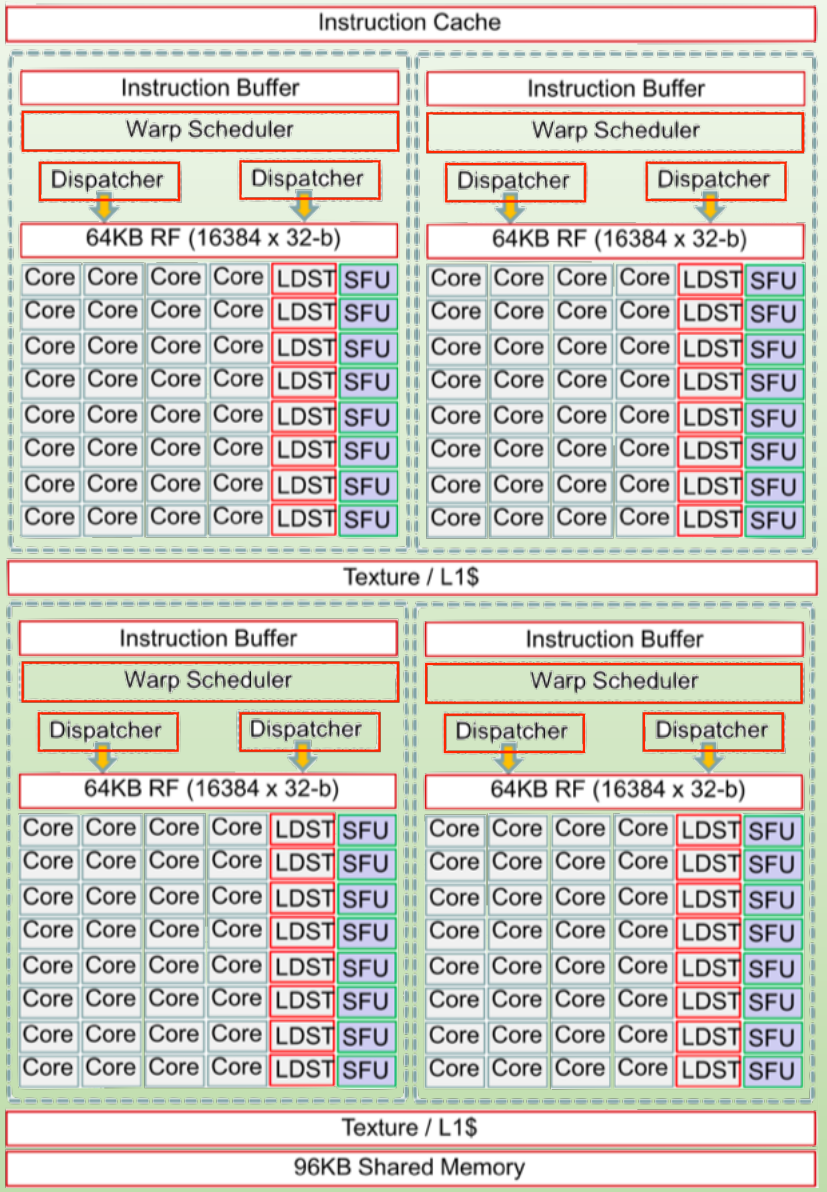
MAXWELL overview

- ✓ Maxwell architecture includes **6 Graphics Processing Clusters** (**GPCs**)
- ✓ each **GPC** houses **4 SMMs** (this is the Maxwell version of SM)
- ✓ the highest-end Maxwell (**GTX Titan X**) houses **24 SMMs** in total
  - each containing **128 cores** for a total of **3072** cores
- ✓ This is barely an increase in the core count as compared to **Kepler's 2880 max**
- ✓ It is clear that the design priority in Maxwell was **not to increase** the **core** count but to make every core more capable by giving them **more resources** inside the **SMM**
- ✓ resources shared by **128 cores+32 SFUs** (192 cores+64DPUs+32SFUs in Kepler)

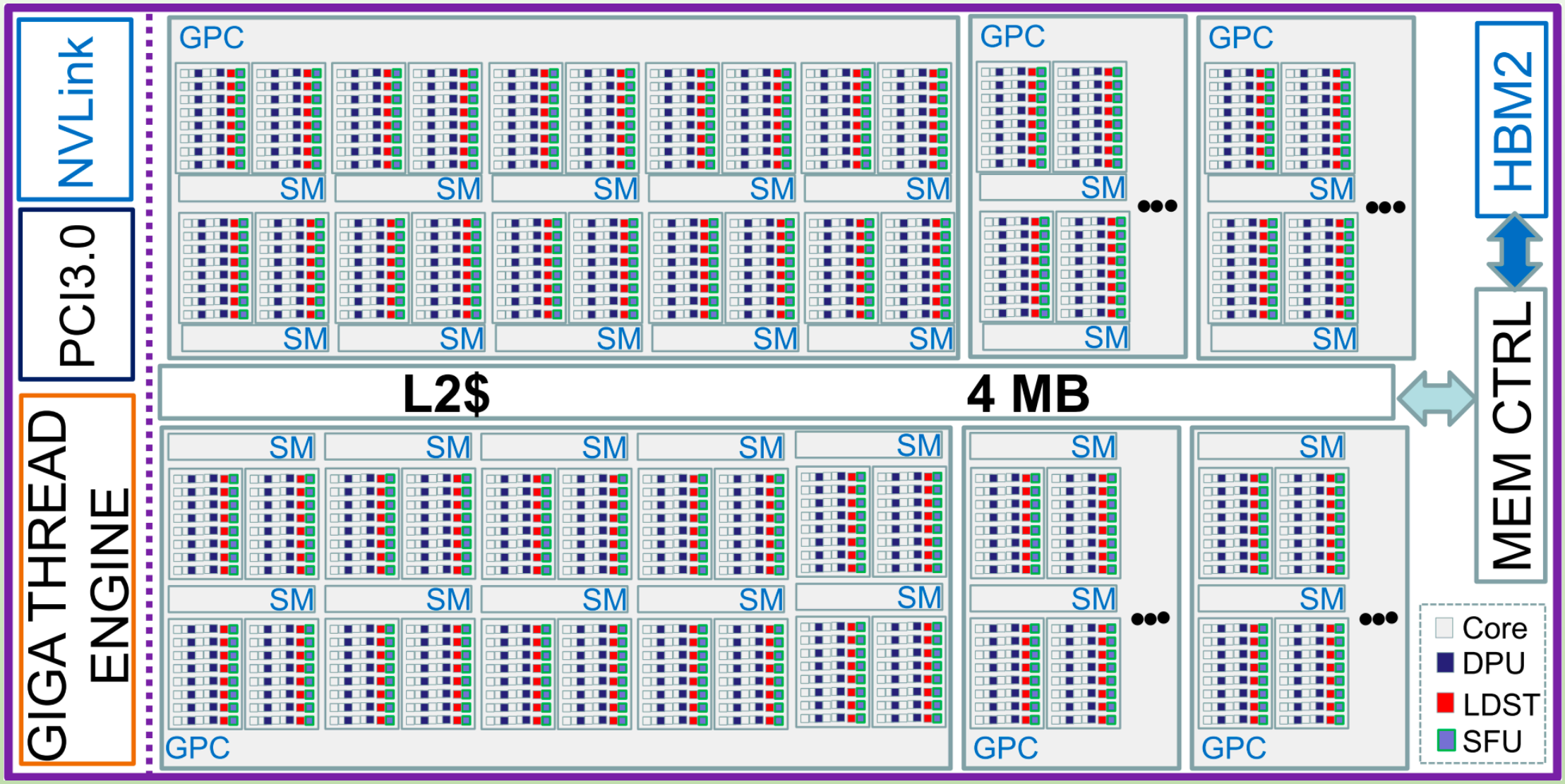
GM200 Maxwell SMM structure

- 4 identical "**sub-units**"
- **sub-units** contain
  - **32 cores**
  - **8 LD/ST** queues
  - **8 SFUs**
  - their own Instruction Buffer
- 2 sub-units share a **96KB shared memory**
- **No DPUs** (double instr. take 32× longer than float instr. in Maxwell)





Architettura PASCAL

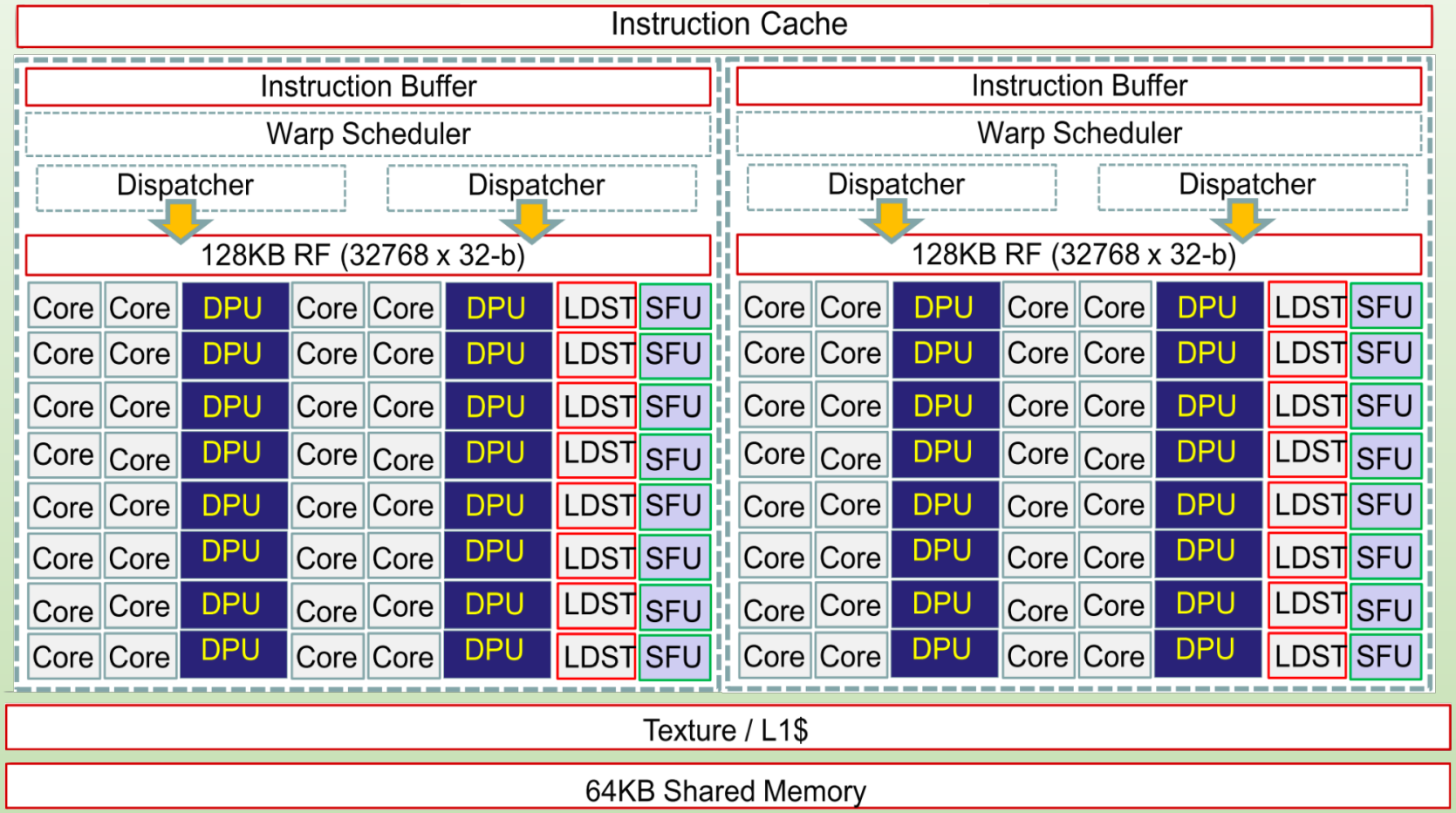


Pascal overview

- ✓ The **L2$** is **4MB** although it is only **2MB** in GTX 1070
- ✓ The total amount of global memory is **12GB** for **Titan X**, **8GB** for **GTX 1070**, and **16GB** for the compute accelerator **P100** (slightly larger than the Kepler and Maxwell generations)
- ✓ **K80** was a two-in-one GPU (containing two K40's) with a total of **24GB** or **32GB**
- ✓ Large amount of global memory helps in **heavy computations** that require the storage of a significant **amount** of **data** inside the GPU (**Deep Learning application** memory is necessary to store the neural nodes)
- ✓ The most drastic change in the Pascal microarchitecture is the support for the emerging **High Bandwidth Memory** (**HBM2**)
- ✓ Using this memory technology is able to deliver **720 GBps** by using its ultra-wide **4096-bit** memory bus (K80 GM bandwidth is 240 GBps for each GPU using a 384-bit bus)
- ✓ New type of **NVLink bus**, which **boasts** a **80 GBps** transfer rate (much faster than the 15.75 GBps the PCIe 3.0 bus and alleviates the CPU←→GPU data transfer bottleneck)

GP100 Pascal SM structure

- **64 cores** per SM
- **16 DPUs**
- **8 LD/ST** units
- **8 SFUs**
- **256 KB (2x128)** registers
- **2048** active threads
- **32 active blocks**



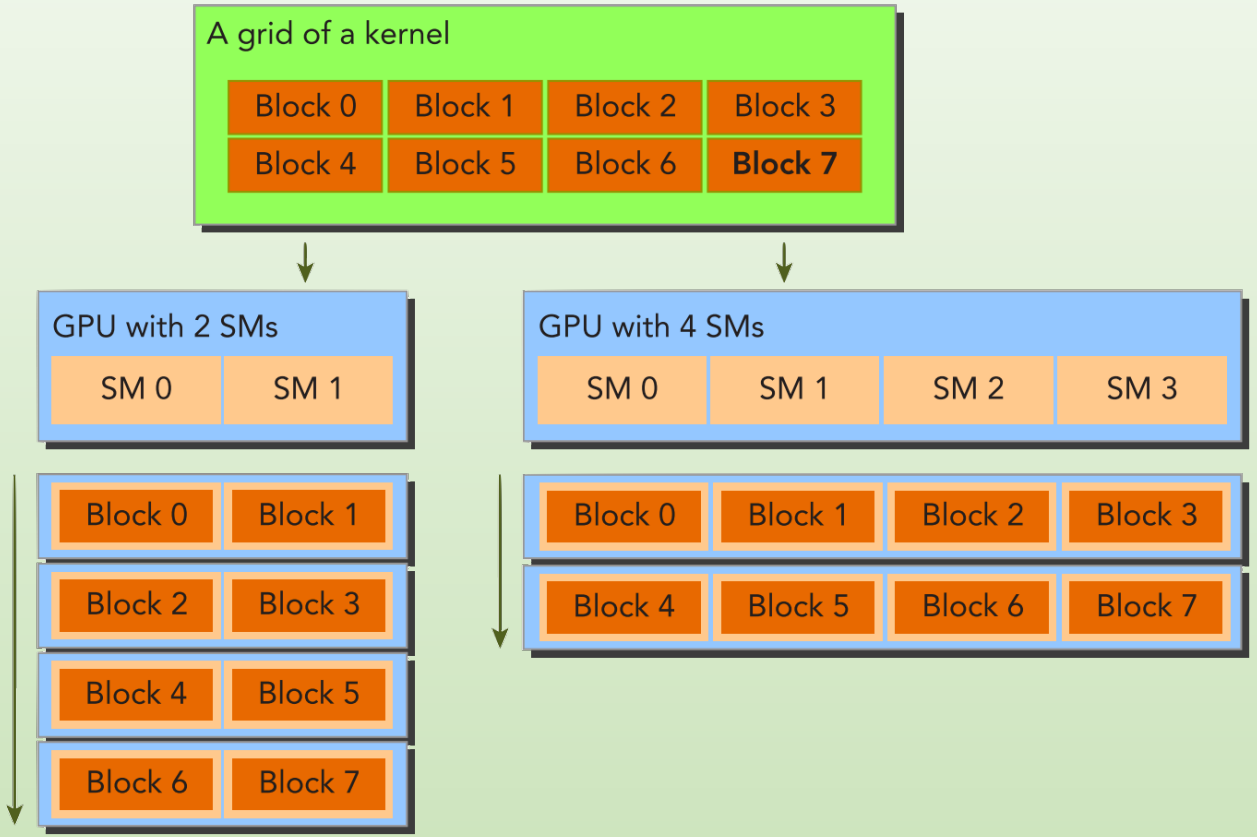
Transparent scalability

Transparent scalability:

- ✓ La capacità di eseguire lo stesso codice applicativo su diverse architetture: numero variabile di compute core e di SM

Vantaggi:

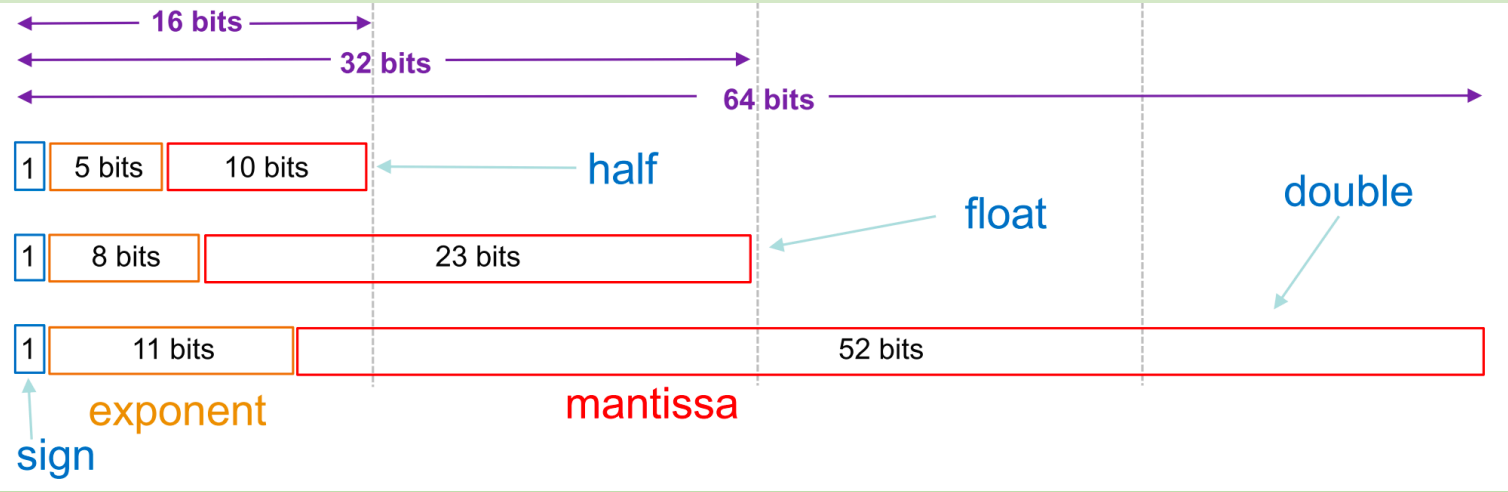
- ✓ **speed-up** di un programma parallelo proporzionale alla dimensione dei dati del problema
- ✓ **Riduce** l'onere dal lato **sviluppo** applicazione e **migliora l'usabilità** delle applicazione
- ✓ Più **importante dell'efficienza**: meglio un programma che scala bene di un programma efficiente ma non scalabile



Senza dover cambiare codice dell'applicazione il kernel con griglia 2 x 4 può essere eseguito indifferentemente sul device con 2 o 4 SM e il tempo scala in accordo alle risorse computazionali disponibili

Double Precision Units (DPU)

- ✓ **How fast**? Without a dedicated **DPU** each core is forced to use its **FPU** to perform double-precision computations in multiple (e.g., 24 or 32) cycles
- ✓ **DPU** is **slightly** physically **larger** than a **core** (VLSI IC design)
- ✓ Physical area required to design a multiplier goes up **quadratically** with the **number** of **bits**
  - single-precision floating point number has **a 23-bit mantissa**
  - double precision has a **52-bit mantissa (**a 4× larger chip area for a DPU vs FPU)



Confronti: Peak GFLOPS e Peak DGFLOPS

✓ Peak **Giga-Floating-Point-Operations** (**GFLOPS**)

$$\text{Peak\_GFLOPS} = f \cdot n \cdot 2$$

f = clock for a CUDA core
n = total number of CUDA cores

- Es. **GTX 780** the core clock is **863MHz** and there are **2304** CUDA **cores**: 863 × 2304 × 2 = **3977 GFLOPS** (single prec.)

✓ Peak **Double-Giga-Floating-Point-Operations** (**GFLOPS**)

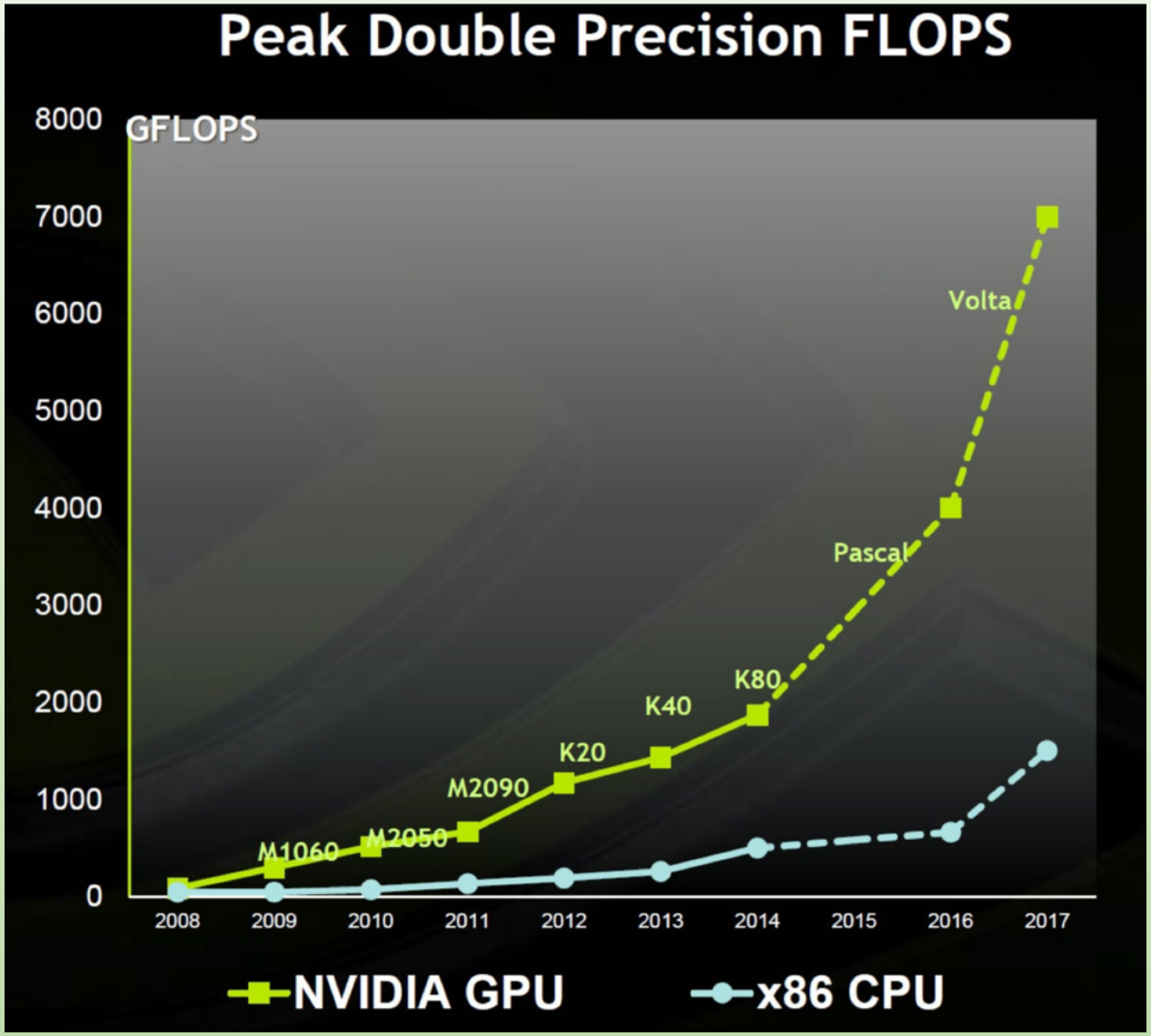
$$\text{Peak\_DGFLOPS} = \begin{cases} \frac{\text{Peak\_GFLOPS}}{24} & \text{Kepler GPUs with no DPU} \\ \frac{\text{Peak\_GFLOPS}}{3} & \text{Kepler GPUs with a DPU} \\ \frac{\text{Peak\_GFLOPS}}{32} & \text{Maxwell GPUS} \\ \frac{\text{Peak\_GFLOPS}}{32} & \text{Pascal GPUS with no DPU} \\ \frac{\text{Peak\_GFLOPS}}{2} & \text{Pascal GPUS with a DPU} \end{cases}$$

- Es. GTX 1070 **Pascal GPU with no DPUs** (1920 cores)
- **GFLOPS** = 1506 × 1920 × 2 = **5783**
- **DGFLOPS** = 5783 ÷ 32 = **181**

Confronti

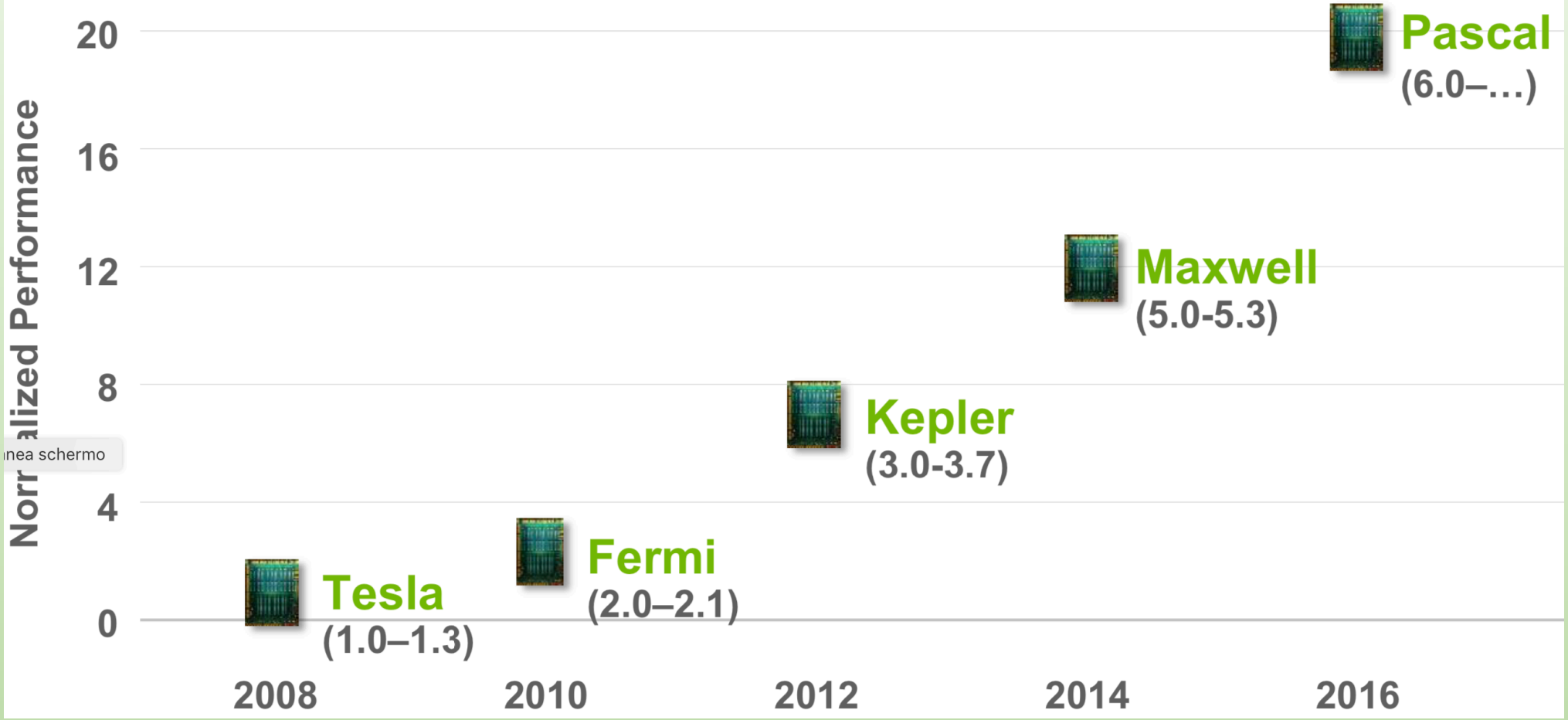
Family	**Engine**	**GTX Model**	**# SMs**	**# cores**	**# DPUs**	**# SFUs**	Peak Compute Power		
							`float`	`double`	Ratio
Fermi	GF110	550 Ti	4	192		16	691		
		GTX 580	16	512		64	1581		
Kepler	GK110	GTX 780	12	2304	0	384	3977	166	24×
	GK110	Titan	14	2688	896	448	4500	1500	3×
	2×GK110	Titan Z	30	5760	1920	960	8122	2707	3×
	2×GK210	K80	26	4992	1664	832	8736	2912	3×
Maxwell	GM200	980 Ti	22	2816	0	704	5632	176	32×
		Titan X	24	3072	0	768	6144	192	32×
Pascal	GP104	GTX 1070	15	1920	0	480	5783	181	32×
	GP102	Titan X	28	3584	0	896	10157	317	32×
	GP100	P100	56	3584	1792	896	9519	4760	2×
Volta									

GPU vs CPU (Double precision)



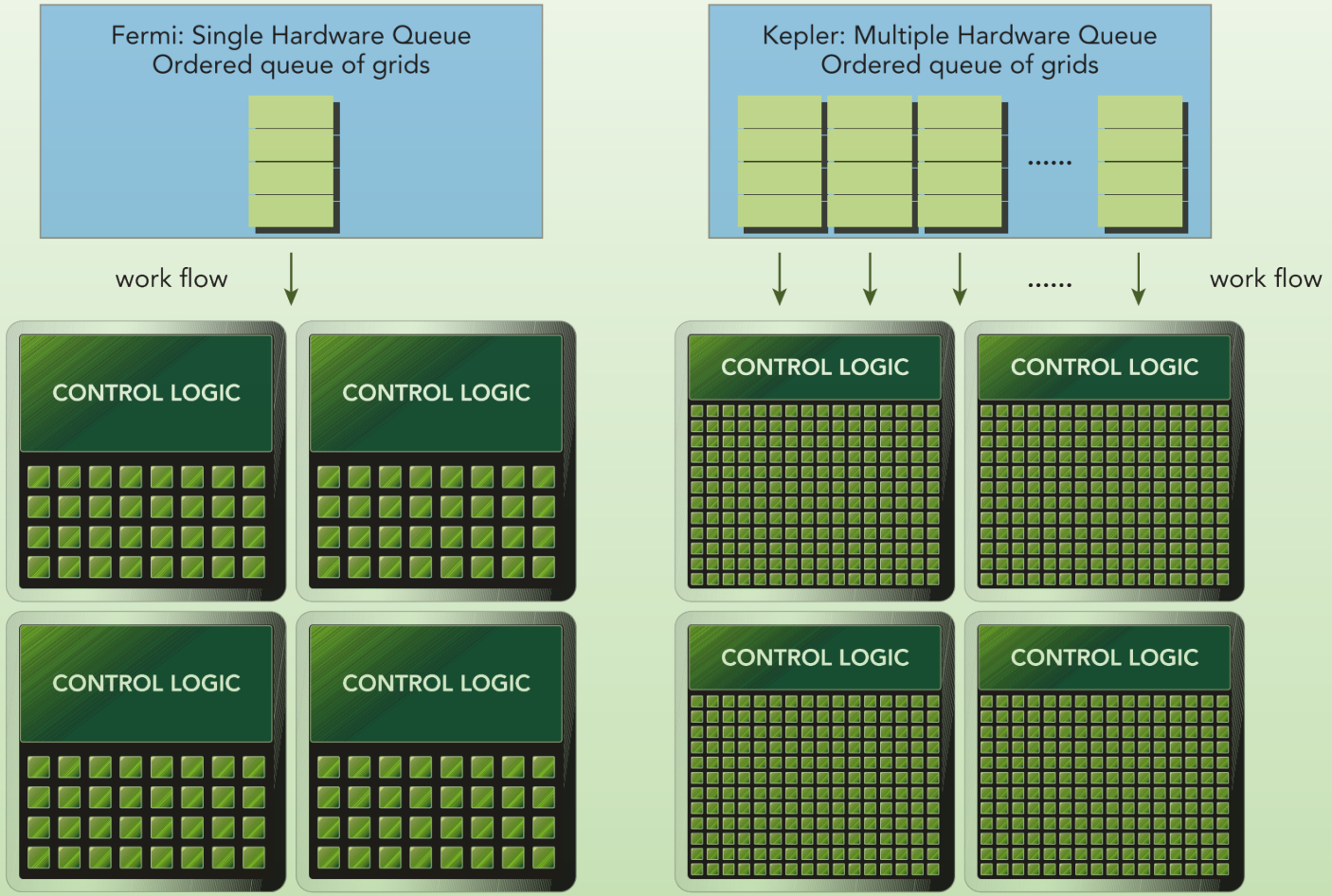
Prestazioni CC Major/Minor

- GPUs are classified according to compute capability with two numbers: CUDA Capability Major/Minor version number



Hyper-Q

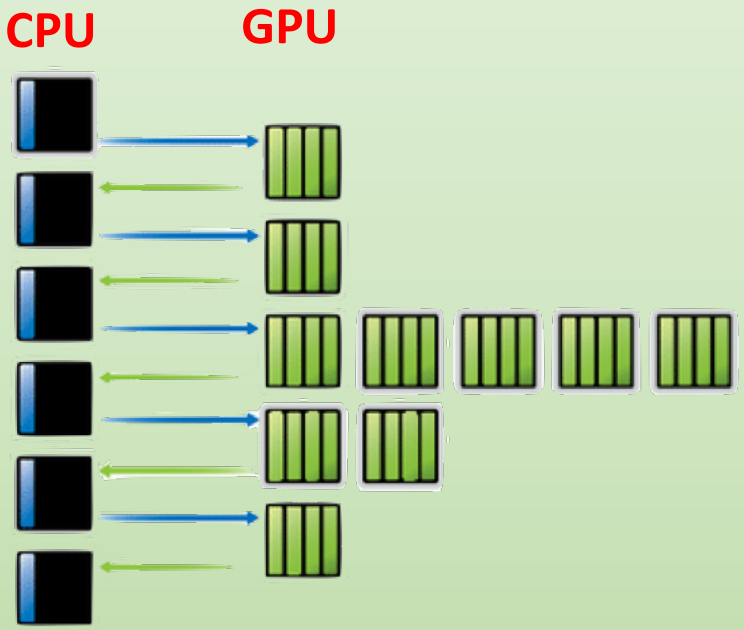
- ✓ Più **connessioni** hardware simultanee tra CPU e GPU
- ✓ Abilita **più CPU core** a lanciare kernel su una singola GPU
- ✓ Abilita i core di CPU ad eseguire **più task simultaneamente** sulla GPU
- ✓ Ci si aspetta di aumentare l'utilizzo della GPU e ridurre l'idle time della CPU
- ✓ Fermi GPUs usano **una sola coda** hardware di lavoro per passare i task dalla CPU alla GPU (un singolo task blocca tutti gli altri task della coda)
- ✓ Kepler Hyper-Q rimuove questa limitazione con **32 code** di lavoro hardware tra host e GPU
- ✓ Hyper-Q attiva molta più **concorrenza** sulla GPU
- ✓ **Massimizza** l'utilizzo della GPU incrementandone le performance in generale

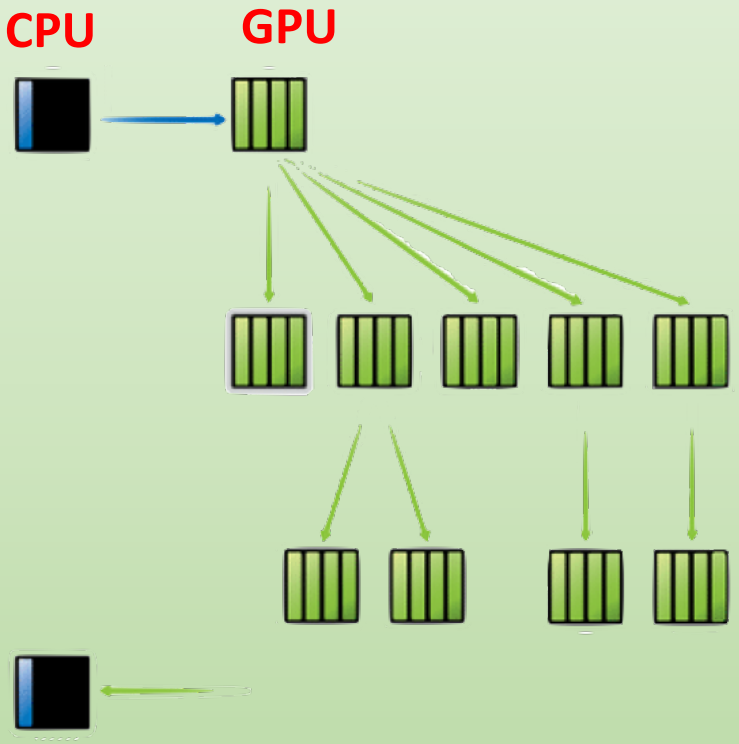


Che cos'è il parallelismo dinamico?

La possibilità di lanciare nuovi kernel (o grid) dalla GPU senza l'intervento della CPU

- dinamicamente…
- simultaneamente…
- indipendentemente…





Dynamic Parallelism

- ✓ Nuova **feature** introdotta nelle GPU Kepler (CC 3.5, CUDA 5.0)
- ✓ Ogni kernel può lanciare un altro kernel e gestire **dipendenze** inter-kernel
- ✓ Gestire la **sincronizzazione** e lo scheduling dei task
- ✓ **Elimina** la necessità di comunicare con la CPU
- ✓ Rende più semplice creare e **ottimizzare** pattern di esecuzione ricorsivi e data-dependent
- ✓ Senza il parallelismo dinamico l'host si occupa di lanciare ogni kernel sulla GPU

Esempio

- ✓ A child **CUDA Kernel** can be called from within a parent CUDA kernel and then **optionally synchronize** on the completion of that child CUDA Kernel
- ✓ The parent CUDA kernel can consume the output produced from the child CUDA Kernel, all **without CPU involvement**

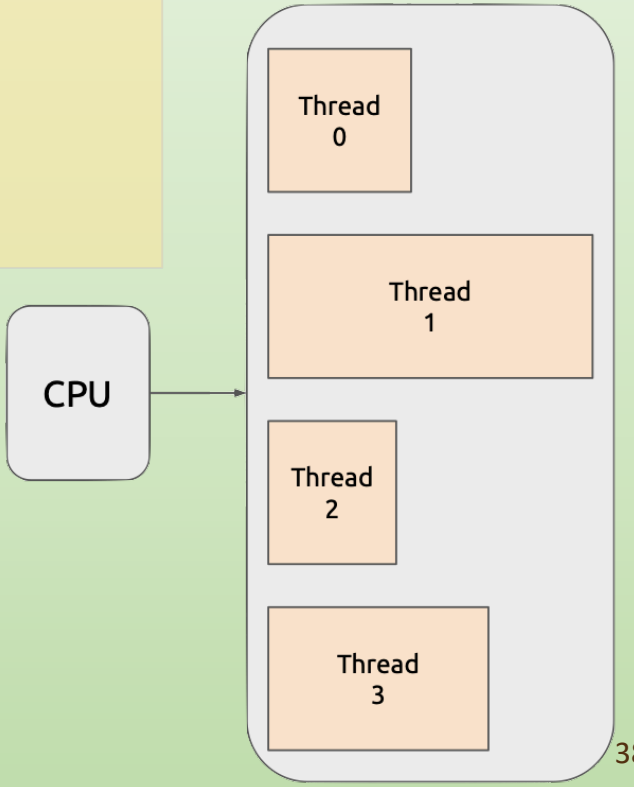
```
__global__ ChildKernel(void* data) {
    //Operate on data
}
__global__ ParentKernel(void *data) {
    ChildKernel<<<16, 1>>>(data);
}
// In Host Code
ParentKernel<<<256, 64>>(data);
```

```
__global__ RecursiveKernel(void* data) {
      if(continueRecursion == true)
            RecursiveKernel<<<64, 16>>>(data);
}
```

Bilanciare il lavoro

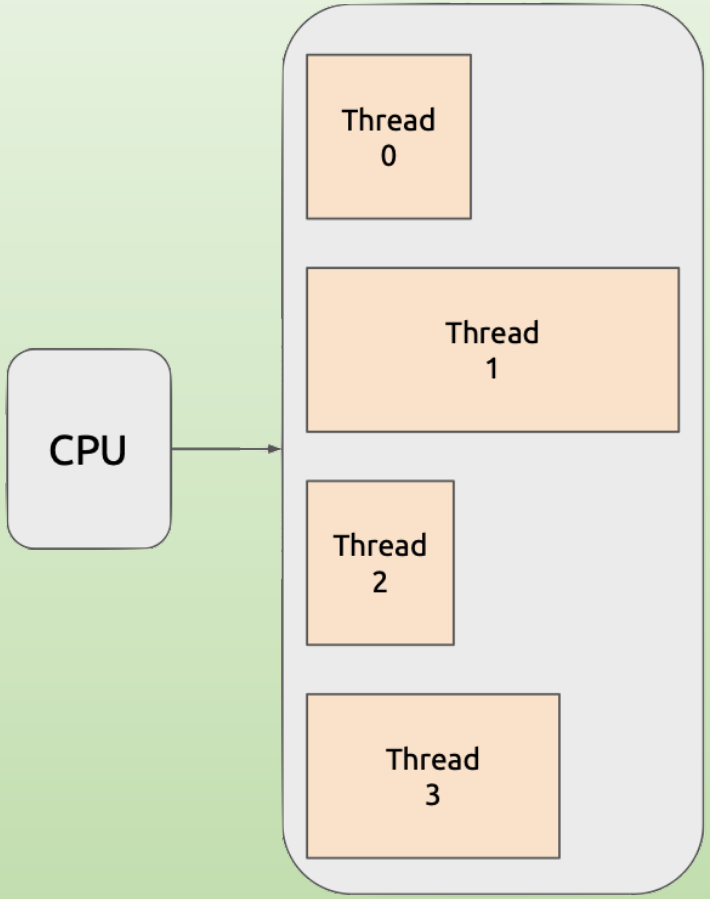
```
__global__ void workDiscoveryKernel(int *starts, int *ends, float *data) {
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        for (int j = starts[i]; j < ends[i]; ++j) {
                process(data[j]);
        }
}
```

- ✓ Il carico di lavoro può dipendere dai dati
- ✓ Thread più ‘pesanti’ impongono tempi di esecuzioni estesi a tutti
- ✓ Spesso si ha uno sbilanciamento del cartico

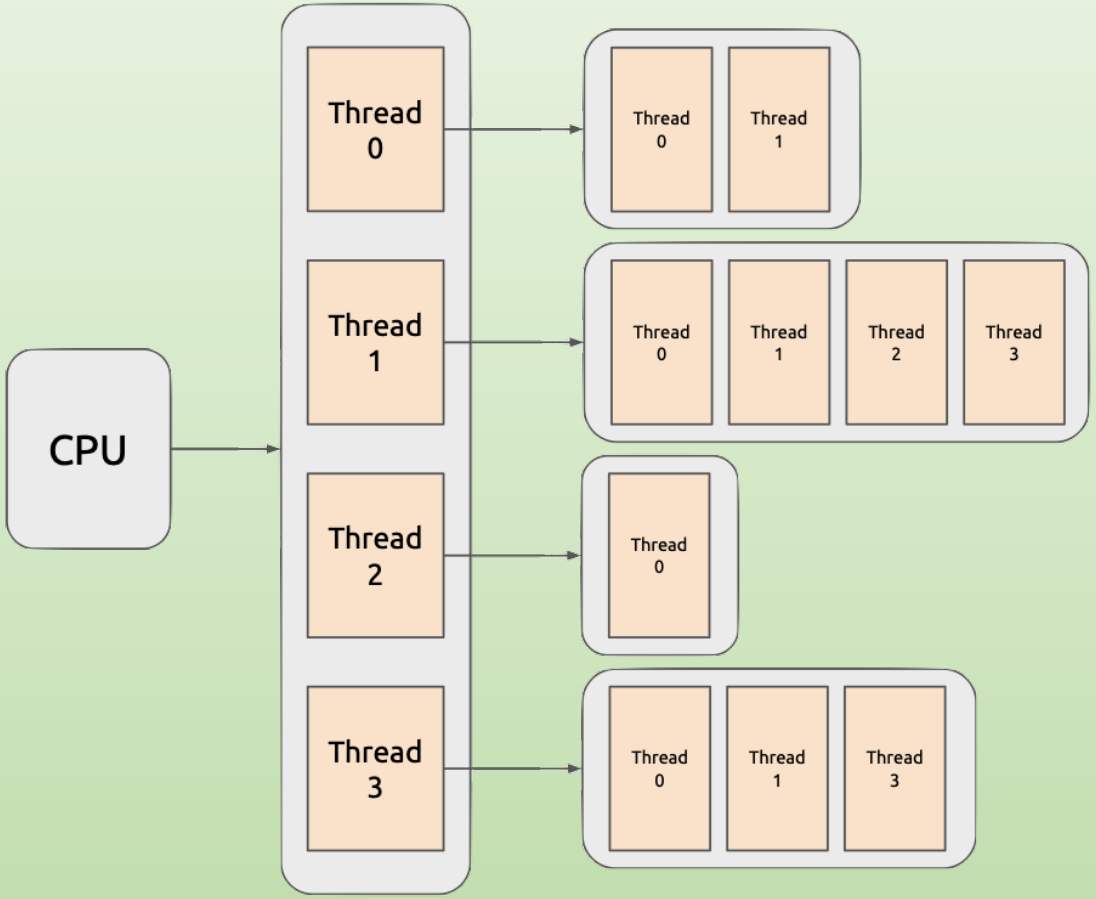


Bilanciare il lavoro

Senza parallelismo dinamico



con parallelismo dinamico

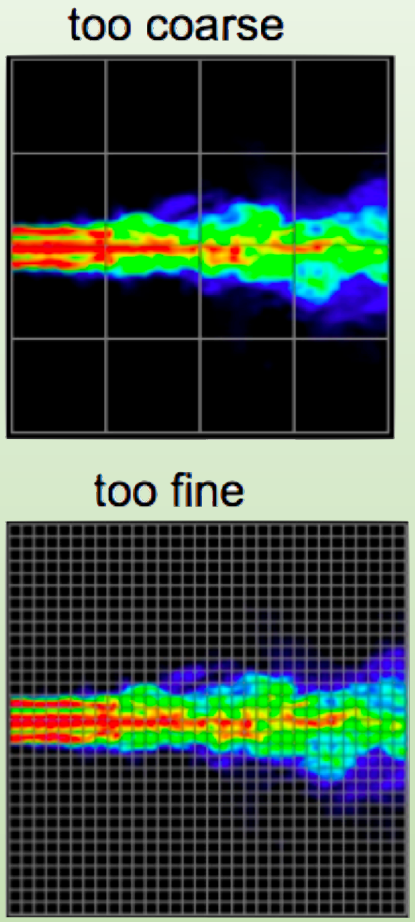


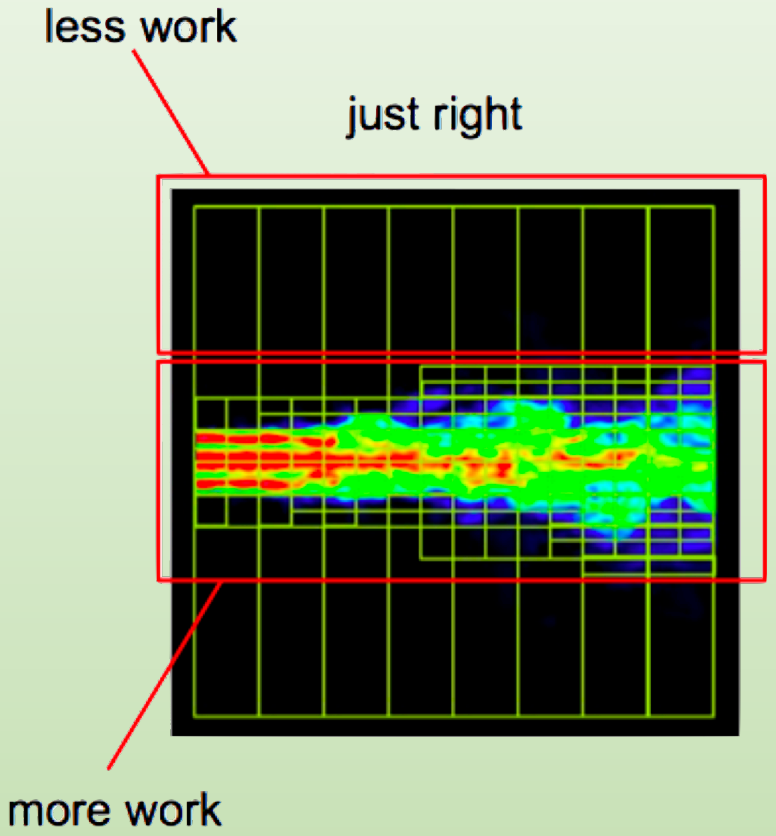
Bilanciare il lavoro

```
__global__ void workDiscoveryKernel(int *starts, int *ends, float *data) {
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        const int N = ends[i] - starts[i];
        workDiscoveryChildKernel<<<(N - 1) / 128 + 1, 128>>>(data + starts[i], N);
}

__global__ void workDiscoveryChildKernel(float * data, int N) {
        int j = threadIdx.x + blockDim.x * blockIdx.x;
        if (j < N) {
                process(data[j]);
        }
}
```

Granularità commisurata al calcolo

- ✓ **Postpone** the **decision** of exactly how many blocks and grids to create on a GPU until **runtime**
- ✓ Taking **advantage** of the GPU hardware **schedulers** and load balancers dynamically
- ✓ Adapting in response to **data-driven decisions** or workloads
- ✓ **Reduce** the need to **transfer execution control** and **data** between the host and device
- ✓ Launch **configuration decisions** can be made at **runtime** by threads executing on the device





https://devblogs.nvidia.com/introduction-cuda-dynamic-parallelism/

Global memory e parall. dinamico

- ✓ Parent and child grids have two points of **guaranteed global memory consistency**
  - When the child **grid** is **launched** by the parent all memory operations performed by the parent thread before launching the child are visible to the child grid when it starts
  - When the child **grid finishes** all memory operations by any thread in the child grid are visible to the parent thread once the parent thread has synchronized with the completed child grid

Memoria locale dei thread

Local memory is private to a thread, and dynamic parallelism is not exception

Child grids have **no privileged access** to the parent thread's local data

global memory data

Kernel KO!

```
__global__ void badParentKernel() {
    float data[10];
    childKernel<<<...>>>(data);
}
```

Kernel OK!

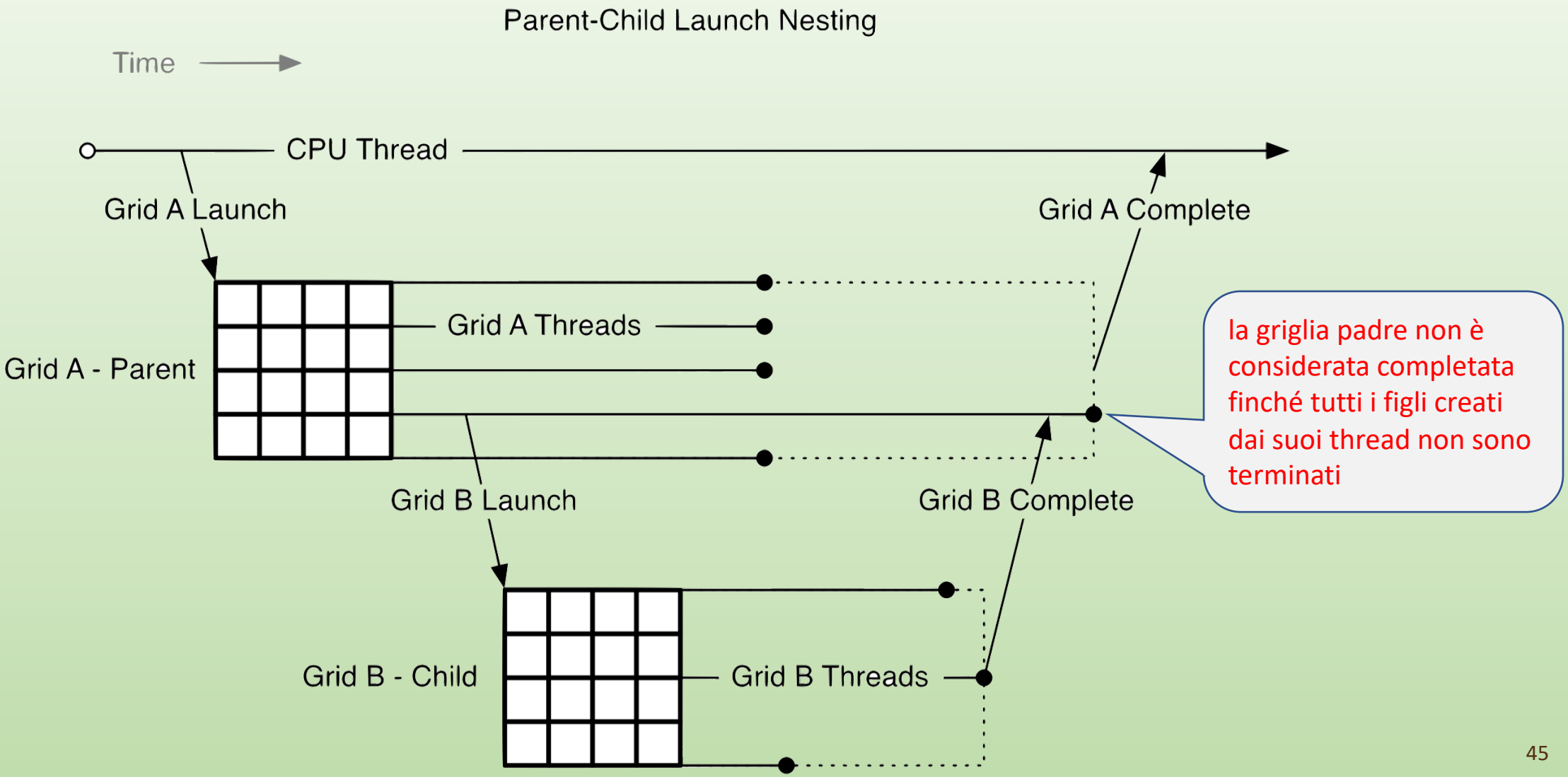
```
__global__ void goodParentKernel(float *data) {
    childKernel<<<...>>>(data);
}
```

Limiti nell'annidamento

- ✓ Note that **every thread** that encounters a **kernel** launch **executes it**
- ✓ If the parent grid has **128** blocks with **64** threads each, and there is no control flow around a child kernel launch, then the grid will perform a total of **8192** kernel launches
- ✓ If you want a kernel to only launch **one child** grid per **thread block**, you should launch the kernel from a single thread of each block

```
// child grid control
if (threadIdx.x == 0) {
   child_k <<< sub_block , sub_grid >>> ();
}
```

sincronizzazione implicita



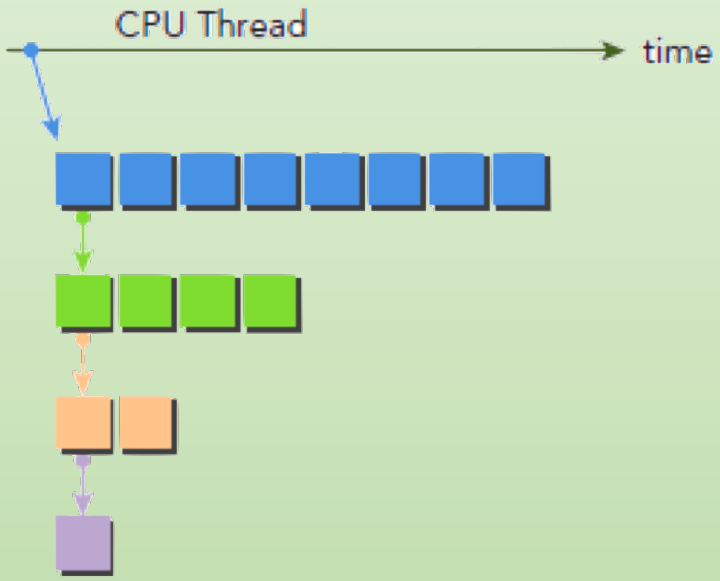
Sincronizzazione esplicita

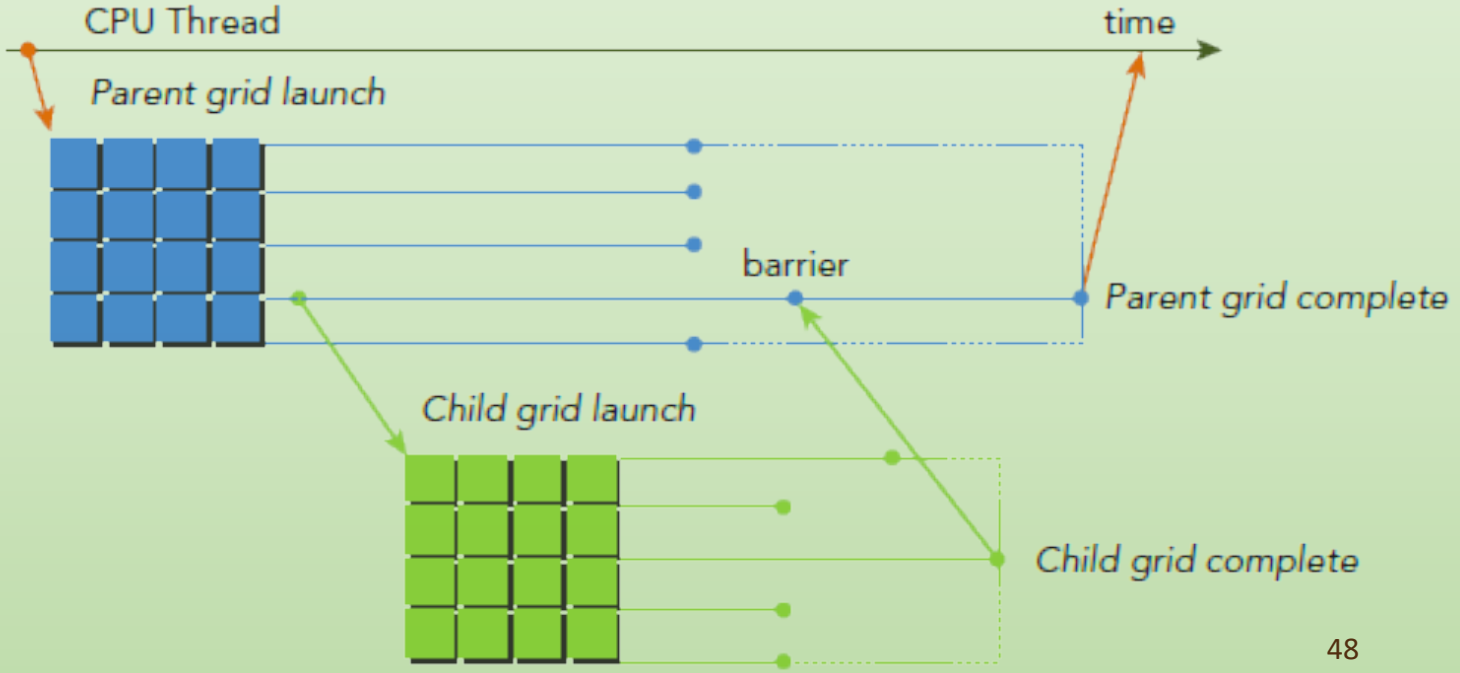
- ✓ the child grid executing **`child_launch`** is only guaranteed to see the modifications to **`data`** made before the child grid was launched
- ✓ Since thread 0 of the parent is performing the launch, the child will be consistent with the memory seen by thread 0 of the parent
- ✓ Due to the first **`__syncthreads`**() call, the child will see **`data[0]=0, data[1]=1, ..., data[255]=255`** (without the **`__syncthreads`**() call, only **`data[0]`** would be guaranteed to be seen by the child).
- ✓ When the child grid returns, thread 0 is guaranteed to see modifications made by the threads in its child grid
- ✓ modifications become available to the other threads of the parent grid only after the second **`__syncthreads`**() call

```
__global__ void child_launch(int *data) {
    data[threadIdx.x] = data[threadIdx.x] + 1;
}

__global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        child_launch<<<1, 256>>>(data);
        cudaDeviceSynchronize();
     }
     __syncthreads();
}

void host_launch(int *data) {
    parent_launch<<<1, 256>>>(data);
}
```

Annidamenti

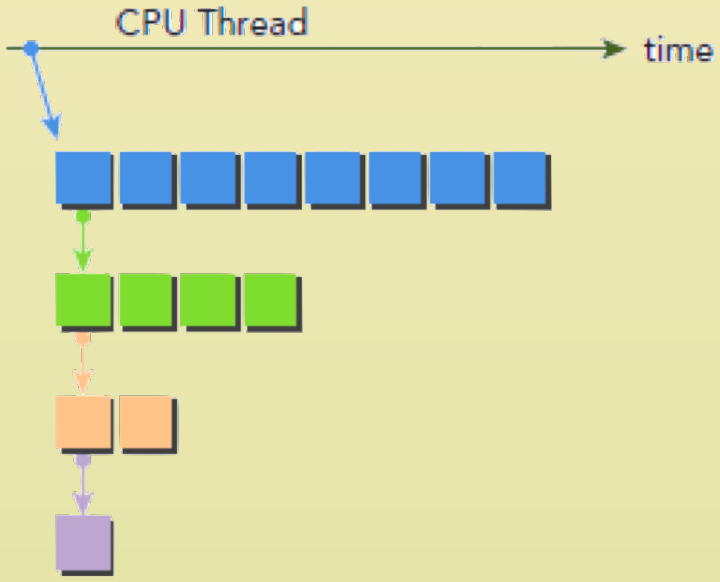
- ✓ Launches are per thread
- ✓ Each thread can launch new kernels
- ✓ Each thread can make nested launches





Nested (recursive) Hello World on the GPU

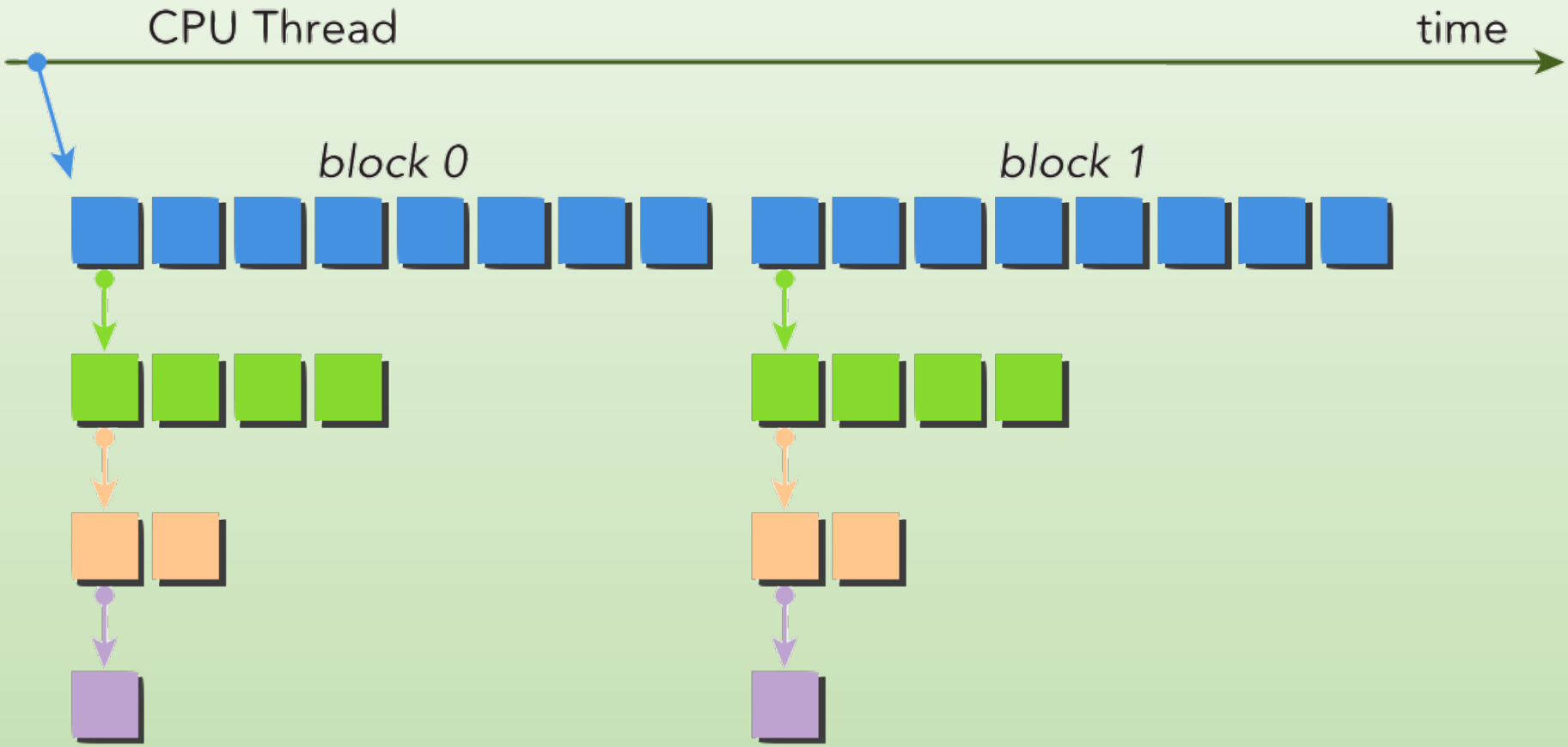
```
__global__ void nestedHelloWorld(int const iSize, int iDepth) {
    int tid = threadIdx.x;
    printf("Recursion=%d: Hello World from thread %d block %d\n", iDepth, tid, blockIdx.x);

    // condition to stop recursive execution
    if (iSize == 1)
        return;

    // reduce block size to half
    int nthreads = iSize >> 1;

    // thread 0 launches child grid recursively
    if (tid == 0 && nthreads > 0) {
        nestedHelloWorld<<<1, nthreads>>>(nthreads, ++iDepth);
        printf("-------> nested execution depth: %d\n", iDepth);
    }
}
```



Esecuzione 1 grid

```
> lab4-nestedHello Execution Configuration: grid 1 block 8
Recursion=0: Hello World from thread 0 block 0
Recursion=0: Hello World from thread 1 block 0
Recursion=0: Hello World from thread 2 block 0
Recursion=0: Hello World from thread 3 block 0
Recursion=0: Hello World from thread 4 block 0
Recursion=0: Hello World from thread 5 block 0
Recursion=0: Hello World from thread 6 block 0
Recursion=0: Hello World from thread 7 block 0
-------> nested execution depth: 1
Recursion=1: Hello World from thread 0 block 0
Recursion=1: Hello World from thread 1 block 0
Recursion=1: Hello World from thread 2 block 0
Recursion=1: Hello World from thread 3 block 0
-------> nested execution depth: 2
Recursion=2: Hello World from thread 0 block 0
Recursion=2: Hello World from thread 1 block 0
-------> nested execution depth: 3
Recursion=3: Hello World from thread 0 block 0
```

Che cosa accade?

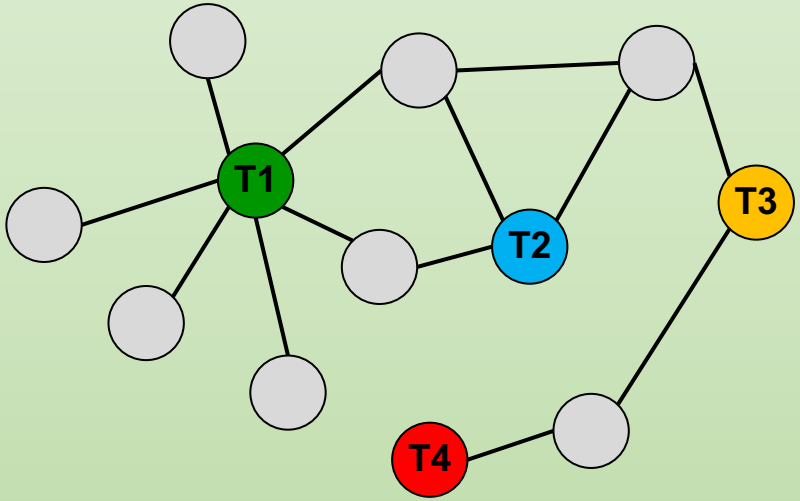


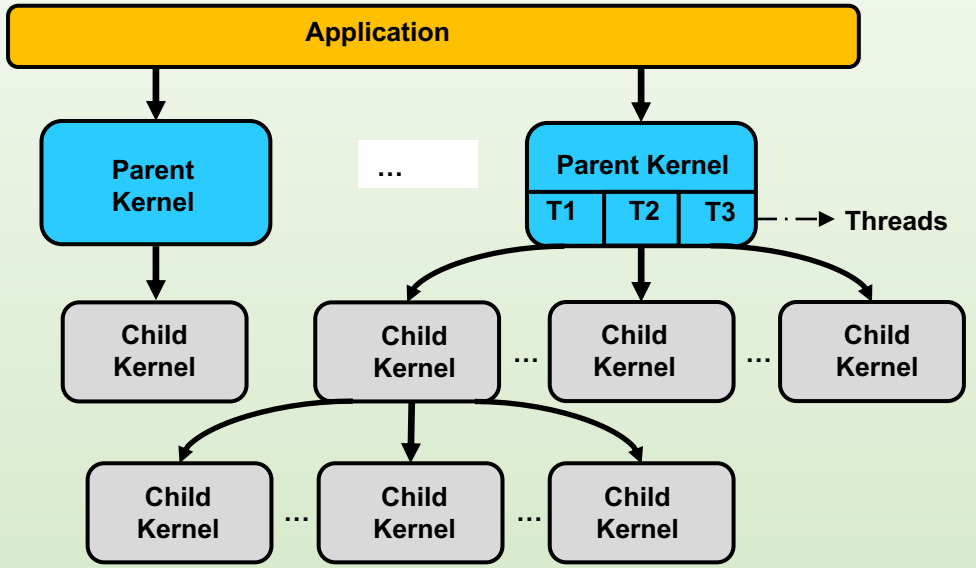
Esecuzione 2 grid

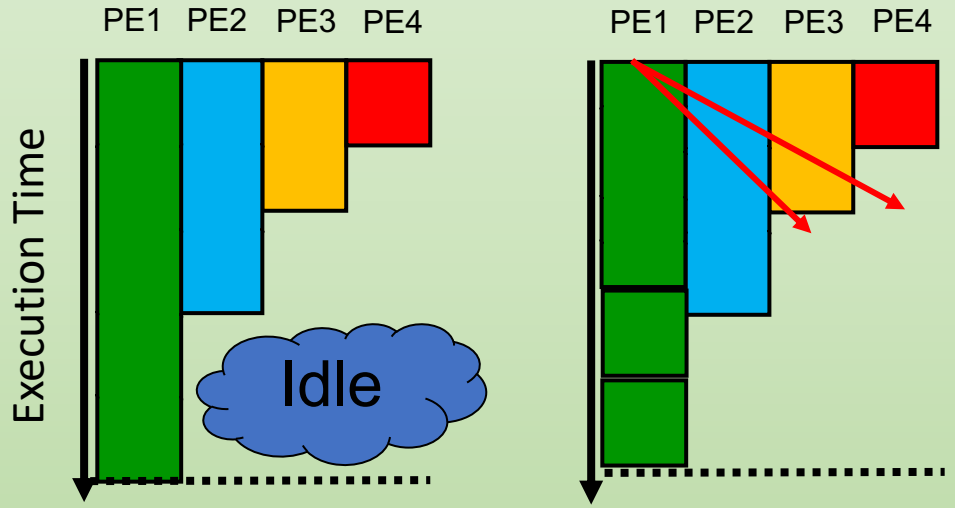
```
> lab4-nestedHello/Release/lab4-nestedHello Execution Configuration: grid 2 block 8
Recursion=0: Hello World from thread 0 block 0
Recursion=0: Hello World from thread 1 block 0
Recursion=0: Hello World from thread 2 block 0
Recursion=0: Hello World from thread 3 block 0
Recursion=0: Hello World from thread 4 block 0
Recursion=0: Hello World from thread 5 block 0
Recursion=0: Hello World from thread 6 block 0
Recursion=0: Hello World from thread 7 block 0
Recursion=0: Hello World from thread 0 block 1
Recursion=0: Hello World from thread 1 block 1
Recursion=0: Hello World from thread 2 block 1
Recursion=0: Hello World from thread 3 block 1
Recursion=0: Hello World from thread 4 block 1
Recursion=0: Hello World from thread 5 block 1
Recursion=0: Hello World from thread 6 block 1
Recursion=0: Hello World from thread 7 block 1
-------> nested execution depth: 1
-------> nested execution depth: 1
Recursion=1: Hello World from thread 0 block 0
Recursion=1: Hello World from thread 1 block 0
Recursion=1: Hello World from thread 2 block 0
Recursion=1: Hello World from thread 3 block 0
Recursion=1: Hello World from thread 0 block 0
Recursion=1: Hello World from thread 1 block 0
Recursion=1: Hello World from thread 2 block 0
Recursion=1: Hello World from thread 3 block 0
-------> nested execution depth: 2
-------> nested execution depth: 2
Recursion=2: Hello World from thread 0 block 0
Recursion=2: Hello World from thread 1 block 0
Recursion=2: Hello World from thread 0 block 0
Recursion=2: Hello World from thread 1 block 0
-------> nested execution depth: 3
-------> nested execution depth: 3
Recursion=3: Hello World from thread 0 block 0
Recursion=3: Hello World from thread 0 block 0
```

Vantaggi

- ✓ Elevata occupazione della GPU
- ✓ Bilanciamento dinamico del carico
- ✓ Supporta ricorsione parallela







Compilazione

As dynamic parallelism is supported by the **device runtime library** kernel must be explicitly linked with **`-lcudadevrt`** on the command line

```
$ nvcc -arch=sm_35 -rdc=true MQDBProd.cu -o MQDBProd -lcudadevrt
```

The flag **`-rdc=true`** forces the generation of relocatable device code a requirement for dynamic parallelism

CUDA Zone…

Uso ottimale delle risorse x SM

Querying device properties

Risponde a domande come: quanti SM sono disponili sul device? Quanti thread x ogni SM? Quanta memoria? ecc.

- ✓ Funzioni delle **API runtime** di CUDA (http://docs.nvidia.com/cuda/cuda-runtime-api)
- ✓ Command-line utility **nvidia-smi** (https://developer.nvidia.com/nvidia-system-management-interface)

Segnatura->

```
int dev_count;
cudaError_t cudaGetDeviceCount(&dev_count);
```

- ➢ Indaga il numero di device disponibili sul sistema

Segnatura->

```
cudaError_t cudaGetDeviceProperties(cudaDeviceProp* prop, int device);
```

- ➢ Restituisce le proprietà del device nella struttura **cudaDeviceProp**

  http://docs.nvidia.com/cuda/cuda-runtime-api/index.html#structcudaDeviceProp.Listing

ESERCITAZIONE:

Usando le API sopra descritte, interrogare i dispositivi presenti ed ottenere:

- Number of CUDA Capable device(s)
- Total amount of global memory
- Maximum number of threads per multiprocessor
- Maximum number of threads per block
- Maximum sizes of each dimension of a block
- Maximum sizes of each dimension of a grid

Determinare quella che ha il massimo numero di Multiprocessors

Esecuzione

Compilazione su lagrange:

```
$ nvcc -arch=sm_20 checkDeviceInfor.cu
```

Esecuzione:

```
$ Detected 3 CUDA Capable device(s)
Device 0: "Tesla M2090"
  CUDA Driver Version / Runtime Version          7.5 / 7.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 5.25 GBytes (5636554752 bytes)
  GPU Clock rate:                                1301 MHz (1.30 GHz)
  Memory Clock rate:                             1848 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
```

Maximum number of threads in the block

- ✓ There are multiple limits. All must be satisfied
  1. The maximum number of threads in the block is limited to 1024. This is the product of whatever your threadblock dimensions are (x*y*z). For example (32,32,1) creates a block of 1024 threads. (33,32,1) is not legal, since 1056 > 1024.
  2. The maximum x-dimension is 1024. (1024,1,1) is legal. (1025,1,1) is not legal.
  3. The maximum y-dimension is 1024. (1,1024,1) is legal. (1,1025,1) is not legal.
  4. The maximum z-dimension is 64. (1,1,64) is legal. (2,2,64) is also legal. (1,1,65) is not legal.
- ✓ Also, threadblock dimensions of 0 in any position are not legal.
- ✓ Your choice of threadblock dimensions (x,y,z) must satisfy *each* of the rules 1-4 above.
- ✓ You should also do proper cuda error checking. Not sure what that is? Google "proper cuda error checking" and take the first hit.

Max Multiprocessors

Stabilire il massimo numero di processori (o qualsiasi atra proprietà) tra le schede esistenti (Es in Lagrange):

Ottenere il numero di device

Ciclo sui device per estrarre informazioni o attivare processi

Ottenere informazioni circa il device

Attivare il device per eseguire operazioni sullo stesso

```
int numDevices = 0;
cudaGetDeviceCount(&numDevices);
if (numDevices > 1) {
        int maxMultiprocessors = 0, maxDevice = 0;
        for (int device=0; device<numDevices; device++) {
                cudaDeviceProp props;
                cudaGetDeviceProperties(&props, device);
                if (maxMultiprocessors < props.multiProcessorCount) {
                        maxMultiprocessors = props.multiProcessorCount;
                        maxDevice = device;
                }
        }
        cudaSetDevice(maxDevice);
}
```

Command-line tool nvidia-smi

Il command-line tool **nvidia-smi** permette di gestire e monitorare le GPU presenti

```
$ nvidia-smi -L
GPU 0: Tesla M2050 (UUID: GPU-e03071f3-6aa4-4351-abed-3dbf96673cd2)
GPU 1: Tesla M2090 (UUID: GPU-25b1e779-cd4a-1bd4-66cd-48d17d2405c3)
GPU 2: Tesla M2090 (UUID: GPU-ad8ac2a3-4256-7174-f7c1-ec56dedaa17c)
```

Per un report completo sul device 0 (MEMORY, UTILIZATION, ECC, TEMPERATURE, POWER, CLOCK, COMPUTE, PIDS, PERFORMANCE, SUPPORTED_CLOCKS, PAGE_RETIREMENT, ACCOUNTING)

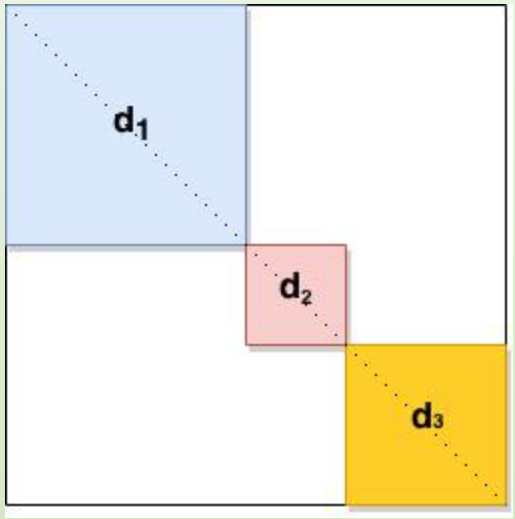
```
$ nvidia-smi -q -i 0
```

Filtrare le info sul device 1

```
$ nvidia-smi -q -i 1 -d UTILIZATION
GPU 0000:11:00.0
    Utilization
        Gpu                               : 0 %
        Memory                            : 0 %
```

```
$ nvidia-smi -q -i 1 -d MEMORY
GPU 0000:11:00.0
    FB Memory Usage
        Total                             : 5375 MiB
        Used                              : 10 MiB
        Free                              : 5365 MiB
```

Challenge: Parallelismo dinamico x MQDB



- ✓ **Specifiche**: una matrice ***A*** in $R^{n \times n}$ di tipo MQDB è composta da ***k*** blocchi $B_1,\ldots,B_k$ ($0 \le k \le n$) disposti sulla diagonale principale. Un esempio con **k=3** è dato in figura. Se $D=\{d_1,\ldots,d_k\}$ denota l'insieme delle dimensioni dei blocchi, allora deve valere: $\sum d_i = n$, pertanto una matrice siffatta è completamente descritta dalla coppia $\langle n,D \rangle$. Vale inoltre che il prodotto di matrici con parametri $\langle n,D \rangle$, è ancora una matrice dello stesso tipo
- ✓ **Input**: due matrici MQDB ***A*** e ***B*** di tipo float descritte da un intero ***n*** e un array ***d[k]***
- ✓ **Output**: la matrice prodotto $C = A * B$ di tipo float

Progetto:

- kernel CUDA con parallelismo dinamico per il prodotto di matrici quadrate diagonali a blocchi