FACULTY OF COMPUTERS AND INFORMATION
CAIRO UNIVERSITY
Established in 1996

Application Acceleration Using the Massive Parallel Processing Power of GPUs

Shady S. Khalifa
Sh.khalfia@fci-cu.edu.eg
Faculty of Computers and Information
Cairo University

Agenda

- Introduction to GPGPU (General Purpose Graphical Processing Units)
- Introduction to NVidia CUDA
- CUDA example: Square matrix multiplication
- Current trends in the GPGPU research (Cloud computing, DBMS, Networks, Security)

Introduction to GPGPU

Central Processing Unit (CPU)[1]

- For more than two decades, **Microprocessors** based on a **sing** drove rapid performance increases in computer applications.

- These microprocessors brought **Giga (billion) floating-point operations per second (GFLOPS)** to the desktop and hundreds of GFLOPS to cluster servers.

- This relentless drive of performance improvement has **allowed** application software to provide **more functionality, have better user interfaces, and generate more useful results.**

- The **users**, in turn, **demand even more improvements** once they become accustomed to these improvements, creating a positive cycle for the computer industry.

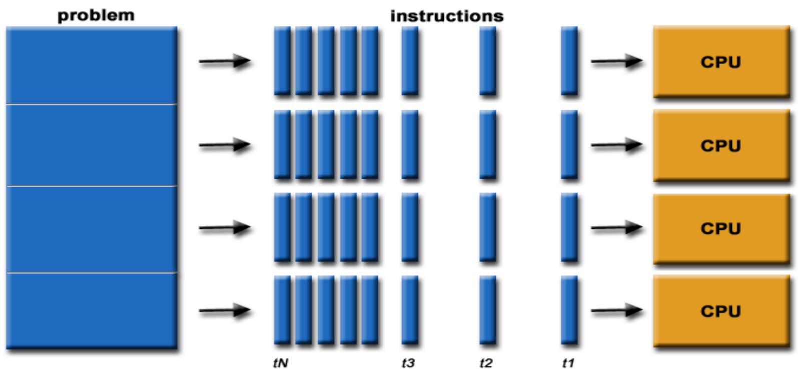
- During the drive, most **software developers have relied on the advances in hardware** to **increase the speed of their applications** under the hood; the same software simply **runs faster as each new generation** of processors is introduced.

- This drive, however, has **slowed** since 2003 due to **energy consumption and heat-dissipation issues** that have **limited** the increase of the clock frequency and the level of productive activities that can be performed in

Multi CPUs (Cores)[1]

- Virtually **all microprocessor vendors** have switched to models where **multiple processing units**, referred to as **processor cores** are used in each chip to increase the processing power.
- Traditionally, the vast majority of **software applications are written as sequential programs**. The execution of these programs can be **understood by a human** sequentially stepping through the code.
- Historically, computer users have become accustomed to the **expectation that these programs run faster with each new generation** of microprocessors. Such expectation is **no longer strictly valid** from this day onward.
- A **sequential program** will only **run on one of the processor cores**, which will not become significantly faster than those in use today.
- **Without performance improvement**, application developers will **no longer be able to introduce new features and capabilities into their software** as new microprocessors are introduced, thus reducing the growth opportunities of the entire computer industry.

Future Applications[1]



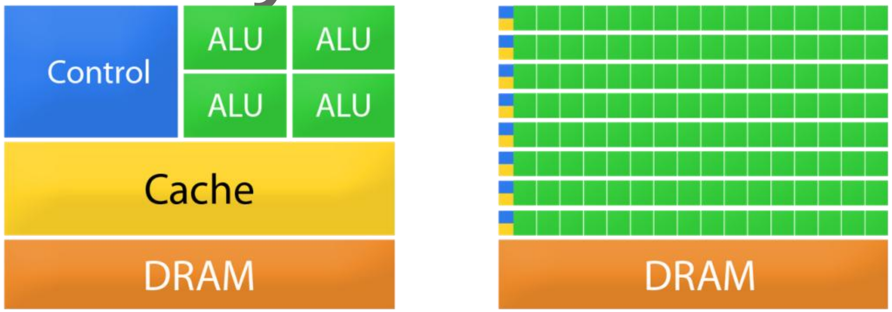
- Applications software that will **continue to enjoy performance improvement with each new generation** of microprocessors will be **parallel programs**, in which **multiple threads** of execution cooperate to complete the work faster.

- The **practice of parallel programming is by no means new**. The high-performance computing community has been developing parallel programs for **decades**. These programs **run on large-scale, expensive computers**.

- Only a **few elite applications can justify the use of these expensive computers**, thus limiting the practice of parallel programming to a small number of application developers.

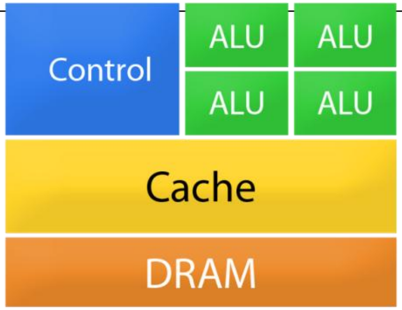
- **Now** that **all new microprocessors are parallel computers**, the number of applications that must be developed as parallel programs has increased dramatically. **There is now a great need for software developers to learn about parallel programming** to cope with the **concurrency revolution.**

Multi Core Vs Many Core [1,2]



	Multi Core	Many Core
Target Programs	Seeks to maintain the execution speed of **sequential programs** while moving into multiple cores	Focuses more on the execution throughput of **parallel programs**
Start	Began as **two-core processors**, with the number of cores approximately **doubling** with **each** semiconductor process **generation**	Began as a **large number of much smaller cores**, and the number of cores **doubles** with each **generation**.
Latest	Intel **Core i7** microprocessor, which has **four** and some models have **six processor cores** .	NVIDIA **GeForce GTX 280** graphics processing unit (GPU) with **240 cores**
Core Capabilities	Each core is an **out-of-order**, **multiple instruction issue** processor implementing the full x86 instruction set, supports hyper threading with **two hardware threads.**	Each Core is a **heavily multithreaded**, **in-order, single-instruction issue** processor that **shares its control and instruction cache with seven other cores.**

CPUs Design Philosophy [1]



- The design of a CPU is **optimized for sequential code performance**.
- It makes use of **sophisticated control logic** to **allow instructions from a single thread** of execution to **execute in parallel or even out of their sequential** order while maintaining the appearance of sequential execution.
- More importantly, **large cache memories** are provided to **reduce the instruction and data access latencies** of large complex applications.
- As of 2009, the new **general-purpose, multicore microprocessors** typically have **four large processor cores** designed to **deliver strong sequential code performance**.

GPUs Design Philosophy [1]

- Shaped by the fast growing **video game industry**, which **requires** the ability to **perform a massive number of floating-point calculations** per video frame in advanced games.

- This demand motivates the GPU vendors to look for ways to **maximize** the **chip area and power budget dedicated to floating point calculations**.

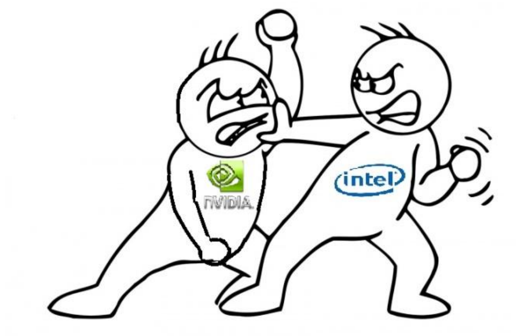
- The **hardware takes advantage** of a **large number of execution threads** to **find work** to do when some of them are **waiting for long-latency memory accesses**, thus **minimizing the control logic** required for each execution thread.

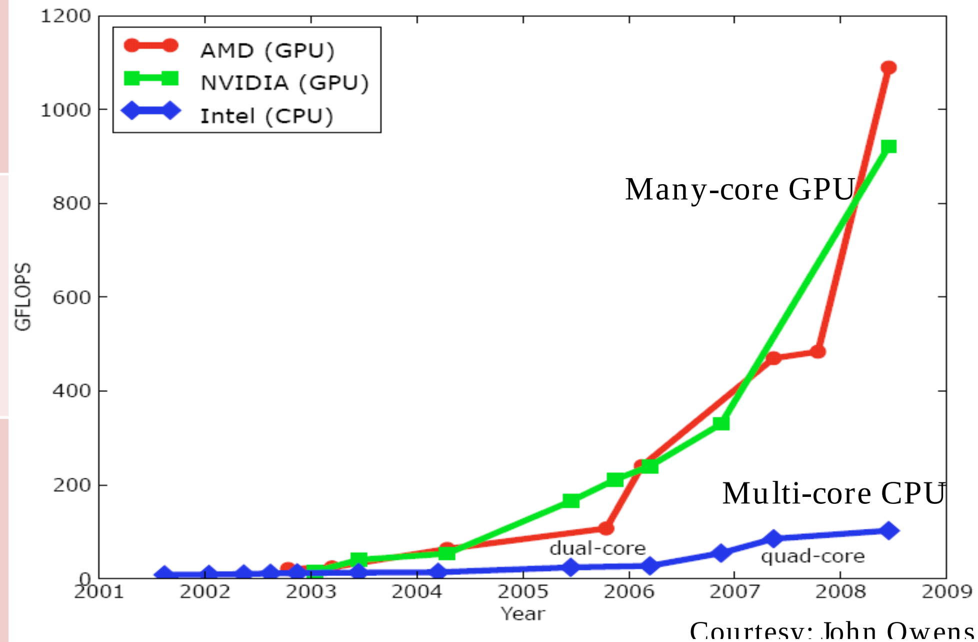
- **Small cache memories** are provided to **help control the bandwidth requirements** of these applications so **multiple threads that access the same memory data do not need to all go to the DRAM**.

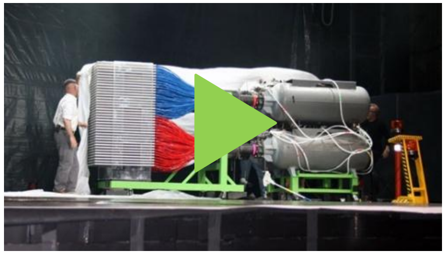
- As a result, much **more chip area** is **dedicated** to the **floating-point calculations.**

CPUs Vs GPUs [1]

	Multi Core (CPU)	Many Core (GPU)
Peak floating-point calculation throughput	100GFLOPS	TFLOPS
Memory Bandwidth (Moving data in and out of DRAM)	50 GB/s	150 GB/s.
Memory bandwidth increase constraints	Have to satisfy requirements from legacy operating systems, applications, and I/O devices.	GPU designers are not constrained as with CPU designers.







GPU IEEE floating-point (IEEE 754) Compliance [1,3,4]

	sign	exponent	fraction
float	1	8	23
double	1	11	52

- An **important consideration** in **selecting a processor** for executing numeric computing applications is the **support for the IEEE floating-point standard**. The standard makes it possible **to have predictable results across processors from different vendors.**

- The standard **defines the way of encoding binary or decimal floating-point numbers** in 64 bits (double precision) and in 32 bits (single precision).

- While support for the IEEE floating-point standard **was not strong in early GPUs**, GPU support for the IEEE floating-point standard has **now become comparable to that of the CPUs**. As a result, one can expect that more **numerical applications will be ported to GPUs** and yield comparable values as the CPUs.

- **Today, a major remaining issue** is that the floating-point arithmetic units of the **GPUs are primarily single precision**. **Applications** that truly **require double-precision floating point were not suitable for GPU** execution.

- **Recent GPUs**, **double-precision execution speed approaches about half that of single precision, a level that high-end CPU cores achieve.** This makes the GPUs suitable for even more numerical applications.

Moving to General Purpose GPU (GPGPUs)

- The large performance gap between sequential and parallel execution has already motivated many applications developers to **move the computationally intensive parts** of their software to **GPUs** for execution.

- In these **computationally intensive parts** there is **more work to do**, there is **more opportunity to divide the work** among **cooperating parallel workers**.

- It should be clear now that **GPUs are designed as numeric computing engines**, and **they will not perform well on some tasks on which CPUs are designed to perform well**; therefore, one should expect that **most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.**

And a lot more in financial analysis, databases and data mining:

http://www.nvidia.com/object/tesla_computing_solutions.html

Introduction to NVidia CUDA

Compute Unified Device Architecture (CUDA) [1,5]

- **Until 2003**, **GPGPU** was **far from easy to program**, even for those who knew graphics programming languages such as OpenGL and Direct3D. **Developers had to map scientific calculations onto problems that could be represented by triangles and polygons**. That's why only a few people could master the skills necessary to use these chips to achieve performance for a limited number of applications. consequently, it **did not become a widespread programming phenomenon**. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent results.

- **In 2003**, a team of researchers led by **Ian Buck unveiled the Brook programming model** to extend C with data-parallel constructs. The **Brook compiler and runtime system exposed the GPU as a general-purpose processor in a high-level language**. Most importantly, Brook programs were not only easier to write than hand-tuned GPU code, they were seven times faster than similar existing code.

- **Nvidia invited Ian Buck to join the company** and start evolving a solution to seamlessly run C on the GPU. Putting the software and hardware together, **Nvidia unveiled CUDA in 2006**, the world's first solution for general-computing on GPUs.

- Nvidia did not represent a change in software alone; **additional hardware was added to the chip area to facilitate the ease of parallel programming.** CUDA programs no longer go through the graphics interface at all. Instead, **a new general-purpose parallel programming interface on the silicon chip serves the requests of CUDA programs**. Moreover, **programmers can use the familiar C/C++ programming tools eliminating the need for using the graphics APIs for computing applications.**

- List of CUDA enabled GPUs : http://developer.nvidia.com/cuda-gpus

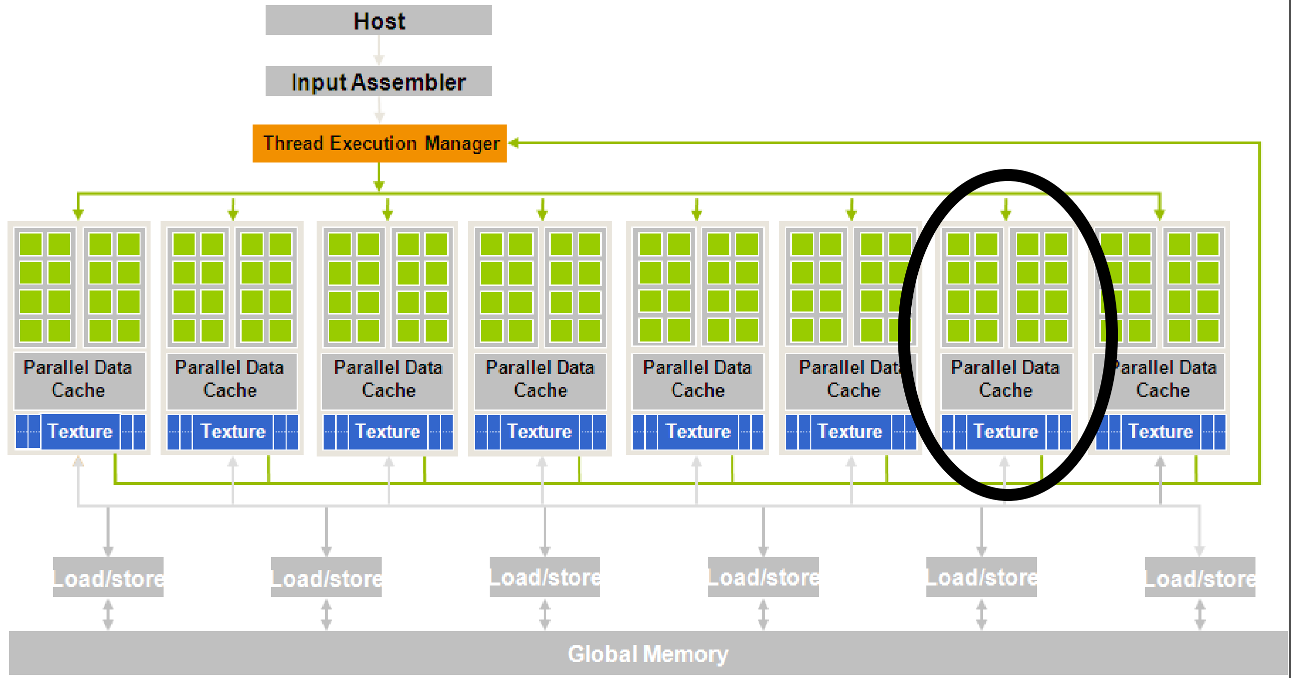
Architecture of Modern CUDA-capable GPUs [1]

CUDA-capable GPU is organized into an array of highly threaded **streaming multiprocessors (SMs).**

Two SMs form a building block.

Each SM has a number of streaming processors (SPs) that **share control logic and instruction cache.**

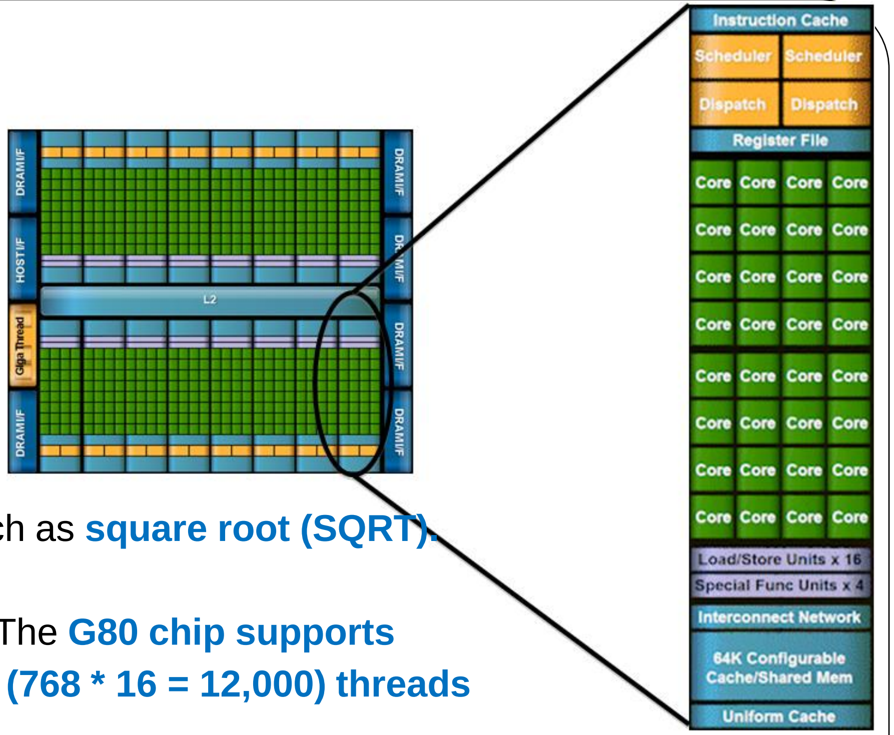
Each GPU currently comes with up **to 4 gigabytes of graphics double data rate (GDDR) DRAM**, referred to as **Global Memory.**



These **GDDR DRAMs differ from the system DRAMs** in that for graphics applications, they hold video images, and texture information, but **for computing they function as very-high-bandwidth, off-chip memory**, though with somewhat **more latency than typical system memory**.

For massively parallel applications, **the higher bandwidth makes up for the longer latency.** GPUs have a **86.4 GB/s of memory bandwidth**, plus an **8 GB/s** (4 GB/s download + 4 GB/s upload) **communication bandwidth with the CPU.**

GPU Power [1]



- The massively parallel G80 chip **has16 SMs, each with 8 SPs (128 SPs in total)** that can support a **total of over 500 GFLOPS**.
- Each SP has a **Multiply–Add (MAD)** unit and an **additional Multiply unit.** In addition, **special function units** perform **floating-point functions** such as **square root (SQRT).**
- While **Intel CPUs support 2 or 4 threads per core**. The **G80 chip supports up to 768 threads per SM**, which **sums up to about (768 * 16 = 12,000) threads**
- The more **recent GT200** consists of **240 SP and supports 1024 threads per SM and up to about 30,000 threads** for the chip.
- Because **each SP is massively threaded**, it can **run thousands of threads per application**. It is **very important to strive for such levels of parallelism when developing GPU parallel computing applications**. A good application typically runs **5000–12,000 threads simultaneously** on this chip.
- In the image: Nvidia Fermi (one of the latest Nvidia inventions) which consists of (16*32) **512 SPs** to give ~**1.5TFLOPS (SP)/~800GFLOPS (DP)**

How to make best benefit from the GPU power [1]

- **It depends** on the **portion** of the application that **can be parallelized, DRAM bandwidth management, on-chip memory capacity management and using CPU to complement the GPU.**

- If the percentage of **time spent** in the part that can be parallelized is **30%**, a **100X speedup of the parallel portion** will **reduce the total execution time by 29.7%.** The **speedup for the entire application** will be only **1.4X.**

- On the other hand, if **99% of the execution time is in the parallel portion**, a **100X speedup will reduce the application execution to 1.99% of the original time.** This **gives the entire application a 50X speedup.**

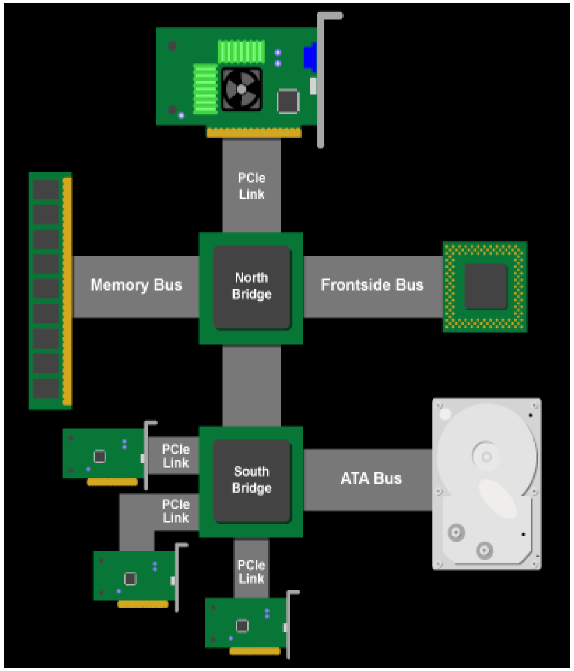
- Therefore, it is **very important** that an application has the **vast majority of its execution in the parallel portion** for a massively parallel processor **to effectively speedup its execution.** This can be **achieved** only after **extensive optimization and tuning of the algorithms.**

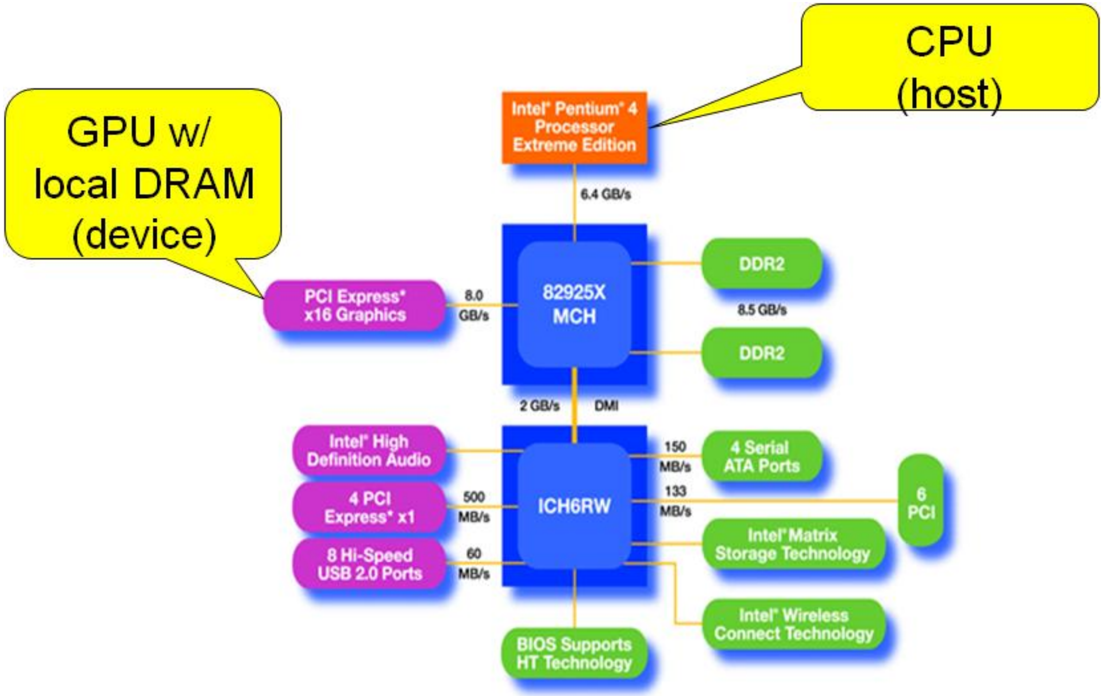
- In general, straightforward parallelization of applications often **saturates the memory (DRAM) bandwidth**, resulting in **only about a 10X speedup**. The **trick is to figure out how to get around memory bandwidth limitations**, which involves doing one of many transformations to **utilize specialized GPU on-chip memories to drastically reduce the number of accesses to the DRAM**. One must, however, **further optimize** the code **to get around limitations such as limited on-chip memory capacity.**

- **In some applications, CPUs perform very well**, making it more **difficult to speed up performance using a GPU. Most applications have portions** that can be **much better executed by the CPU.** Thus, **one must give the CPU a fair chance** to perform and **make sure that code is written in such a way that GPUs complement CPU execution**.

Introduction to CUDA [1]

- The computing system consists of:
  - **Host:** a traditional **CPU**
  - One or more **Devices:** massively parallel processors **GPU**

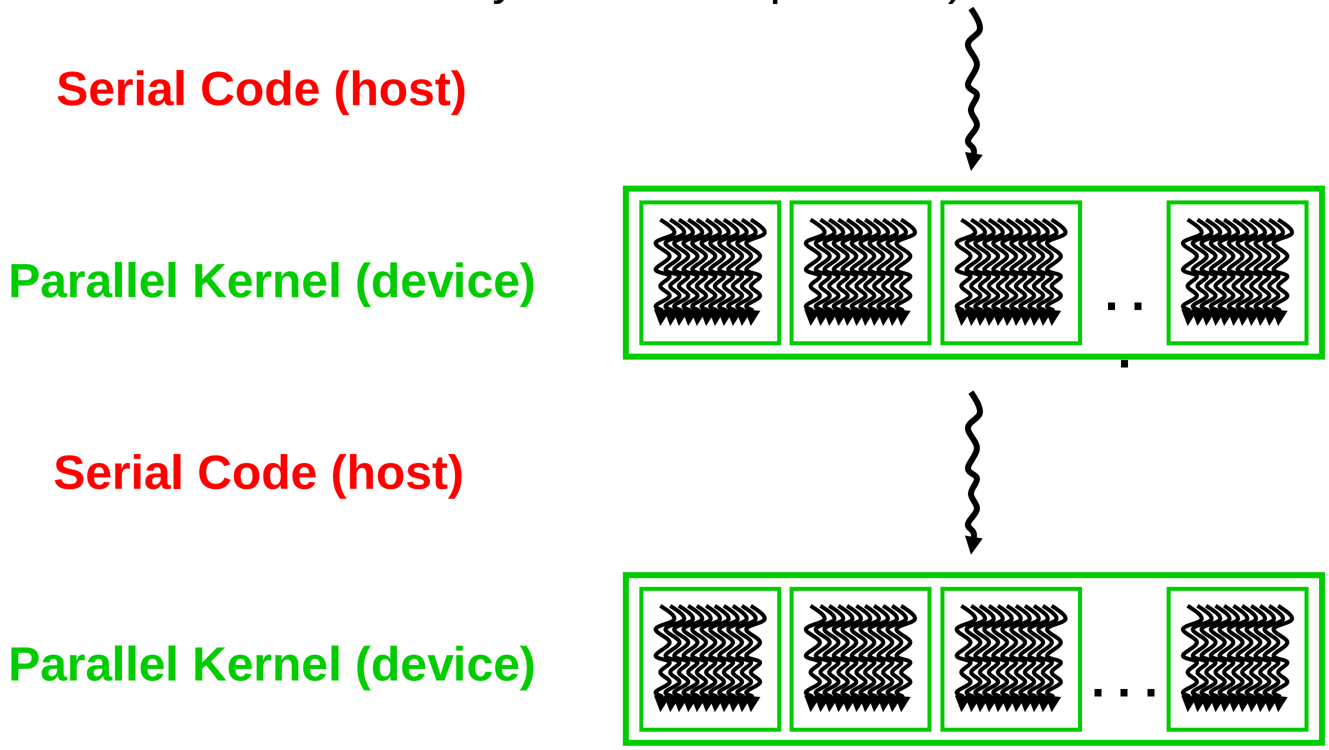




Integrated (Host + Device) C application program [1]

- **Serial** or modestly parallel parts written in **Host C code** and **run on the CPU**

- **Highly parallel parts** written in **Device SPMD (single program, multiple data) kernel C code** and **run on the GPU** (Has its own device memory DRAM and Runs many threads in parallel)



Function Declaration [1]

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- The **__global__** keyword indicates that the function being declared is a CUDA **kernel function**. The function will be **executed on the device** and can only be **called from the host** to generate a grid of threads on a device. **Must return void**. Calls to kernel functions are **Asynchronous**.

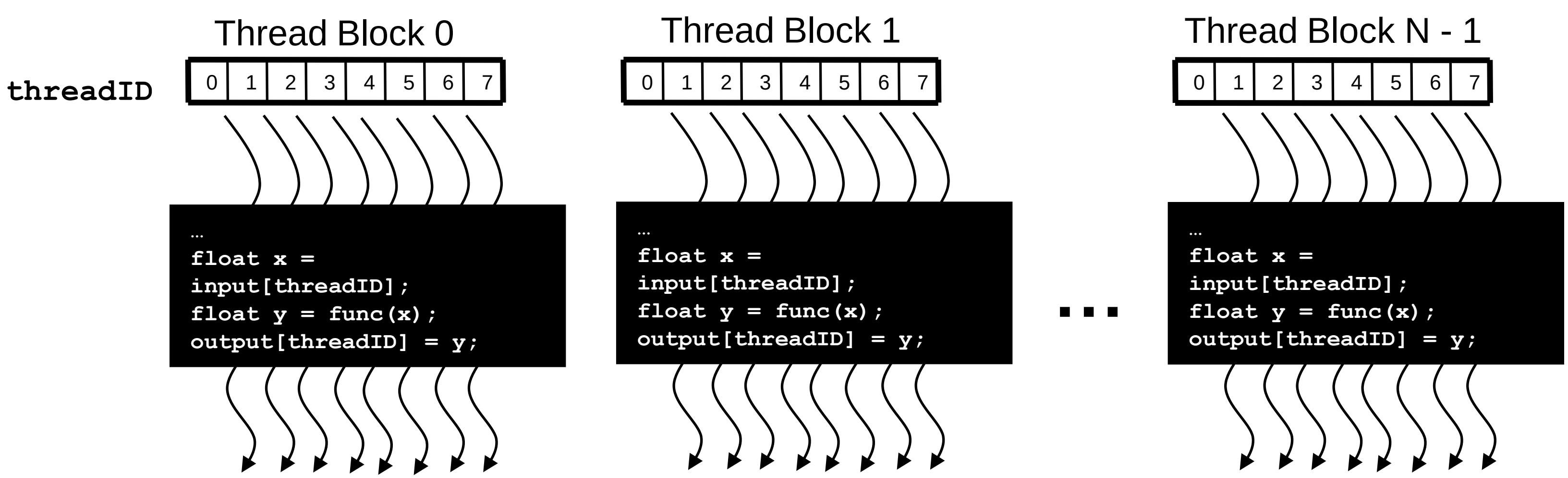
- The **__device__** keyword indicates that the function being declared is a CUDA **device function**. A device function **executes on a CUDA device** and can only **be called from a kernel function** or another device function. Device functions can **NOT have recursive function calls , static variable declaration, variable number of arguments nor indirect function calls through pointers in them**.

- The **__host__** keyword indicates that the function being declared is a CUDA **host function**. A host function is simply a **traditional C function that executes on the host** and can only be **called from another host function**.
  - **By default, all functions** in a CUDA program **are host functions** if they do not have any of the CUDA keywords in their declaration. This makes sense, as many CUDA applications are ported from CPU-only execution environments.

- Both **__host__** and **__device__** **can be used at the same time** in a function declaration. This combination triggers the compilation system to **generate two versions of the same function. One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device** or kernel function. This supports a common use when the same function source code can be simply recompiled to generate a device version.

Thread Blocks [1]

- Divide monolithic thread array into multiple blocks, each of which is defined by a **Block ID**
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in **different blocks cannot cooperate**
- **All threads in all blocks run the same code** (SPMD)
- Each thread has a **Thread ID** that it uses to **compute memory addresses** and make control decisions



Block IDs and Thread IDs [1]

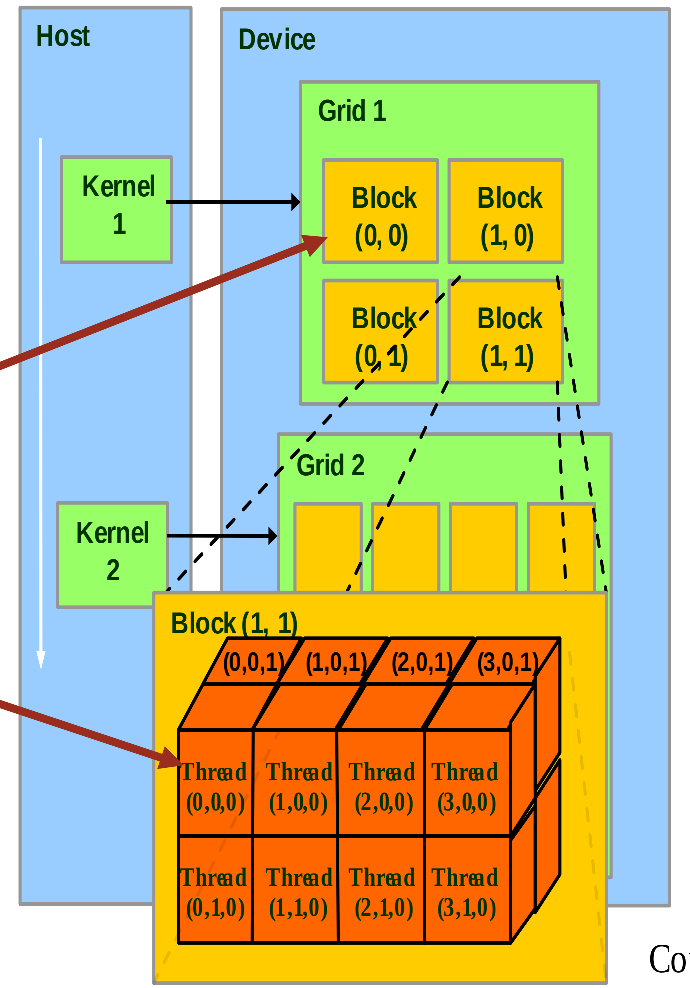
• When a **kernel is invoked**, it is **executed as grid of parallel Threads.** Each **grid is comprised of thousands to millions of lightweight GPU threads.**

• **Threads in a grid are organized into a two-level hierarchy:**

- At the **top level**, **each grid** consists of one or more thread **blocks**. **All blocks in a grid have the same number of threads.** Each block has a **unique two** dimensional **coordinate** given by the CUDA specific keywords ***blockIdx.x*** and ***blockIdx.y.***

- **Each** thread **block** is, in turn, organized as **a three-dimensional array of threads** with a **total size of up to 512 threads.** The coordinates of threads in a block are **uniquely** defined by **three** thread indices: ***threadIdx.x***, ***threadIdx.y***, and ***threadIdx.z***. Not all applications will use all three dimensions of a thread block.

• In the Figure, each thread block is organized into a **4*2*2 three-dimensional array of threads**. This gives **Grid 1 a total of 4*16 = 64 threads.**

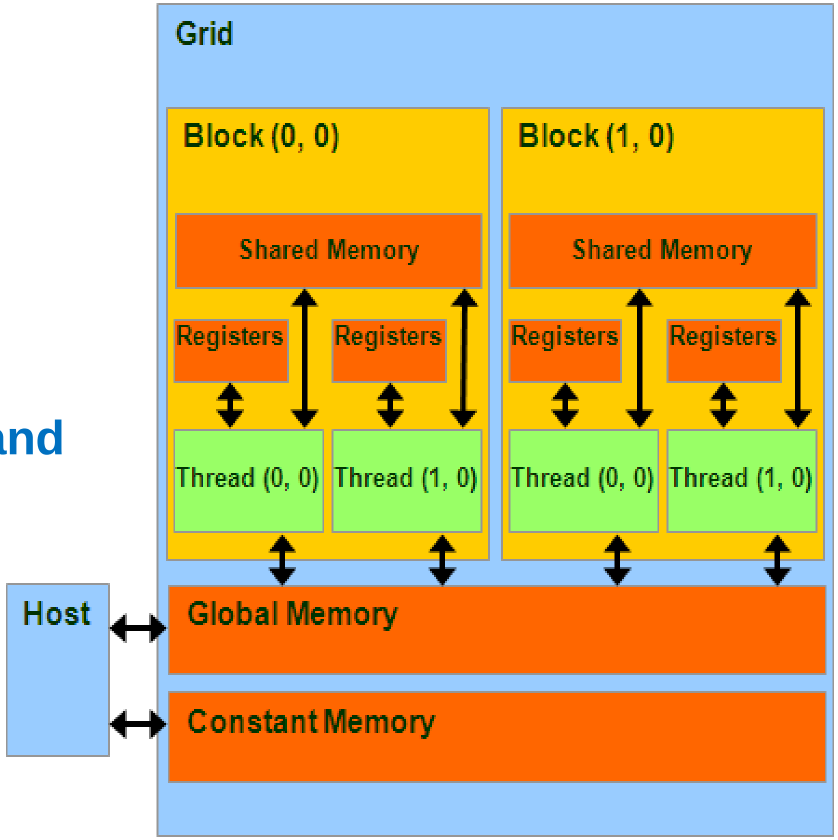


Memories [1]

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

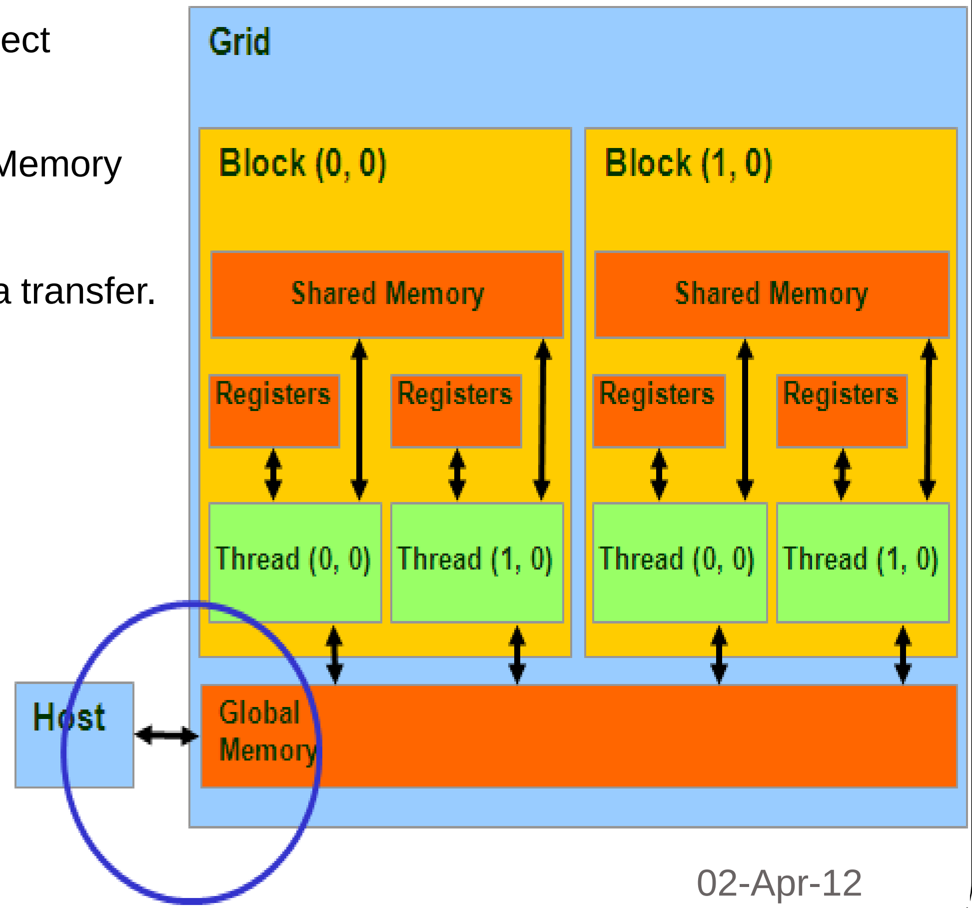
- **Device (Kernel) code can:**
  - R/W per-thread **registers**
  - R/W per-thread **local memory**
  - R/W per-block **shared memory**
  - R/W per-grid **global memory**
  - Read only per-grid **constant Memory**

- **Host code can:**
  - Transfer data to/from per-grid **global and constant memories**

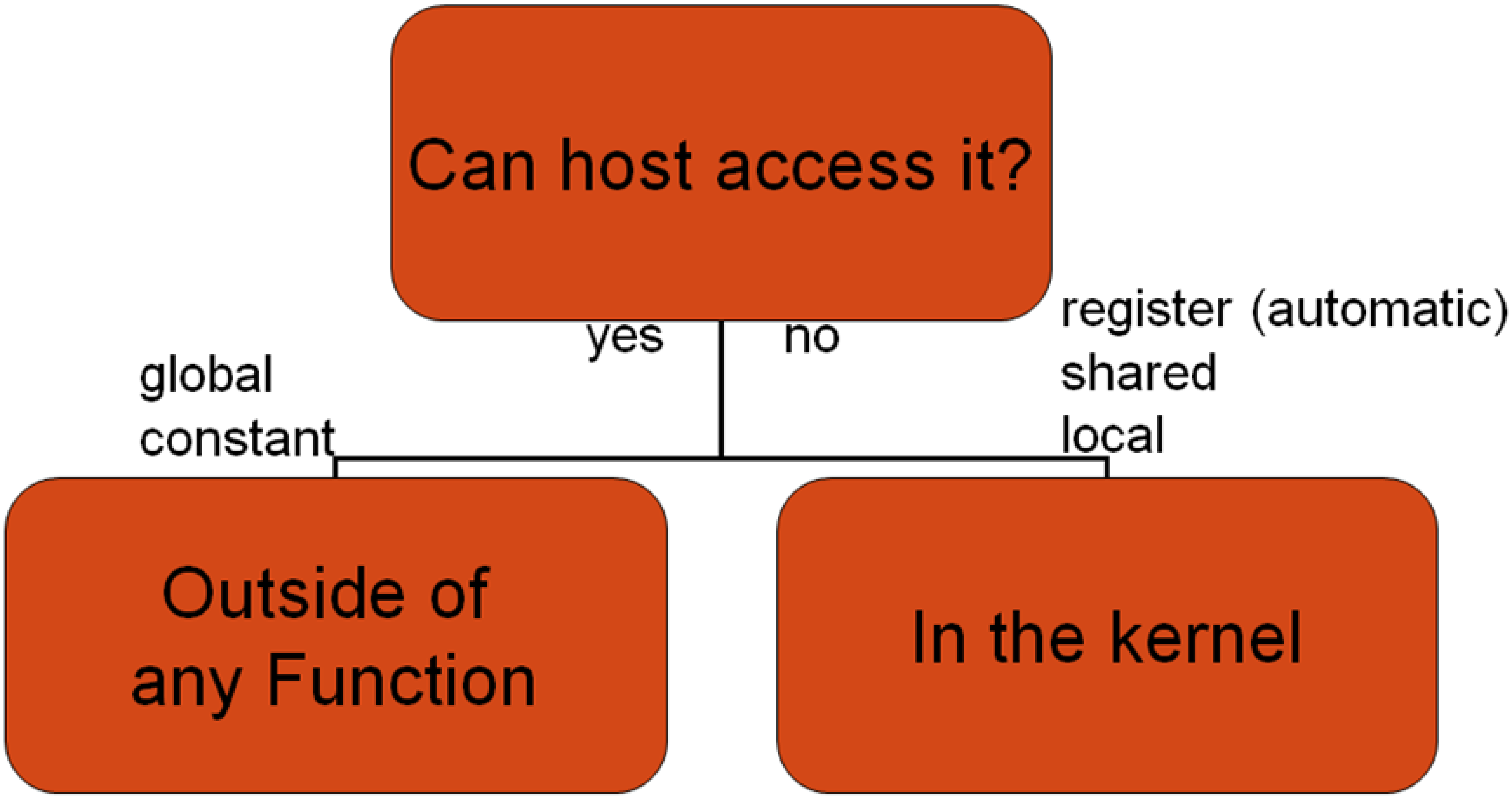


Global Memory [1]

- **Global memory** is the **main means of communicating** R/W Data between **host** and **device**
  - Contents **visible to all threads**
  - **Long latency access**
  - **cudaMalloc():** Allocates object in the device **Global memory.**
    Requires two parameters:
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object
  - **cudaFree()**: Frees object from device Global Memory
    Requires **pointer to freed object.**
  - **cudaMemcpy(): Asynchronous** memory data transfer.
    Requires four parameters:
    - **Pointer to destination**
    - **Pointer to source**
    - **Number of bytes copied**
    - **Type of transfer**
      - o Host to Host
      - o Host to Device
      - o Device to Host
      - o Device to Device



Where to declare variables? [1]



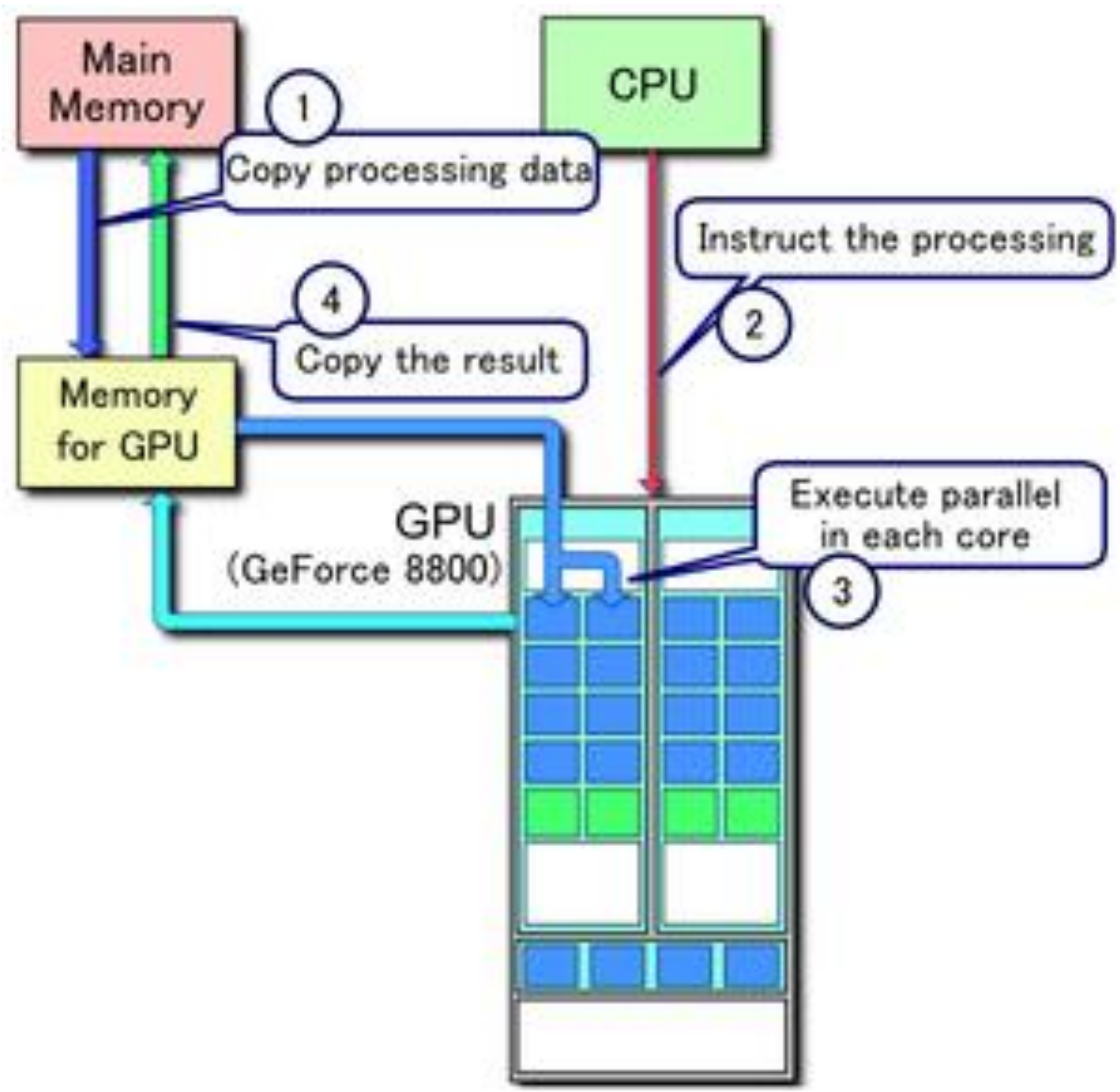
A Common Programming Strategy [1]

- **Global memory** resides in device memory (DRAM) is **much slower access than shared memory**.
- So, a profitable way of performing computation on the device is to **tile data** to take advantage of fast shared memory:
  - **Partition** data into **subsets that fit into shared memory.**
  - **Handle each data subset with one thread block** by:
    - **Loading** the **subset from global memory to shared memory**, using multiple threads to exploit memory-level parallelism.
    - **Performing the computation on the subset from shared memory**; each thread can efficiently multi-pass over any data element
    - **Copying results from shared memory to global memory.**

- **Constant memory** also resides in device memory (DRAM) and is **much slower access than shared memory** but **cached** which can **provide highly efficient access for read-only data.**

- **Carefully divide data according to access patterns:**
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)

Processing flow on CUDA [1]

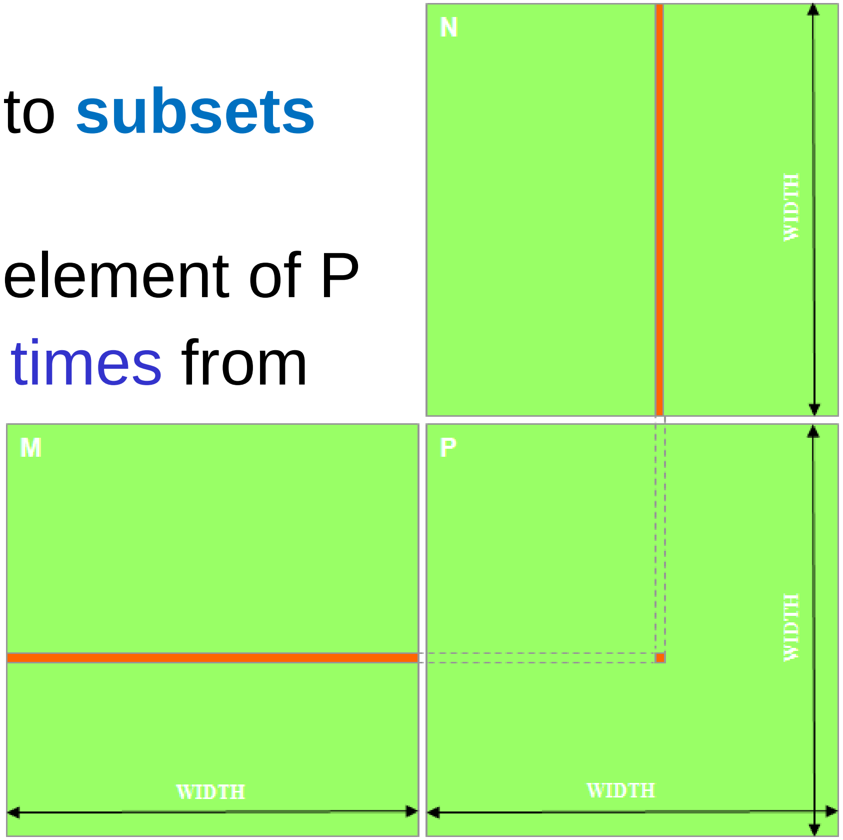


CUDA example: Square matrix multiplication

Example: Square Matrix Multiplication [1]

- P = M * N
- sizeOf(P) = *WIDTH* x *WIDTH*

- Without tiling (**Partition** data into **subsets that fit into shared memory**):
  - One thread calculates one element of P
  - M and N are loaded *WIDTH* times from global memory



Memory Layout of a Matrix in C [1]

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

M

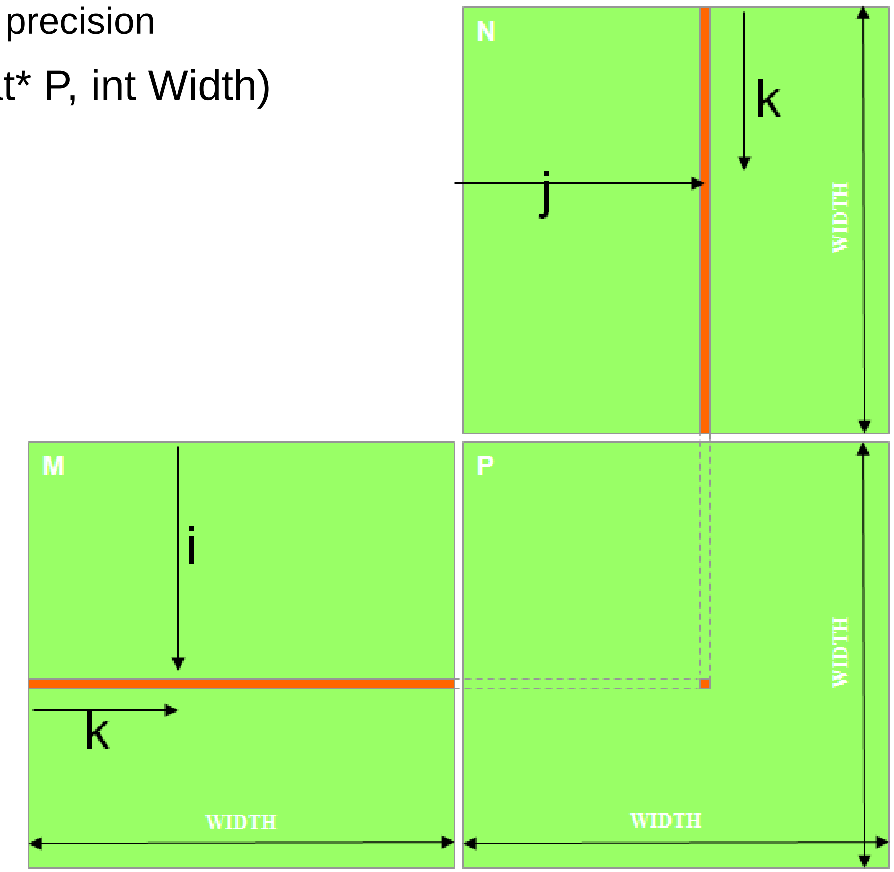
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

For Example: M(i,k) = M(2,3) => M(i * Width + k) = M(2*4+3) = M(11)

Step 1: Matrix Multiplication
A Simple Host Version in C

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Step 2: Input Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  …
1. // Allocate and Load M, N to device memory
   cudaMalloc(&Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

   // Allocate P on the device
   cudaMalloc(&Pd, size);
```

Step 3: Output Matrix Data Transfer (Host-side Code)

2. // Kernel invocation code – to be shown later in Step 5

   …

3. // Read P from the device

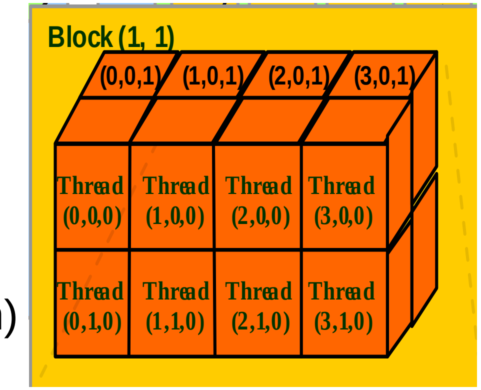
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

   // Free device matrices

   cudaFree(Md); cudaFree(Nd); cudaFree (Pd);

   }

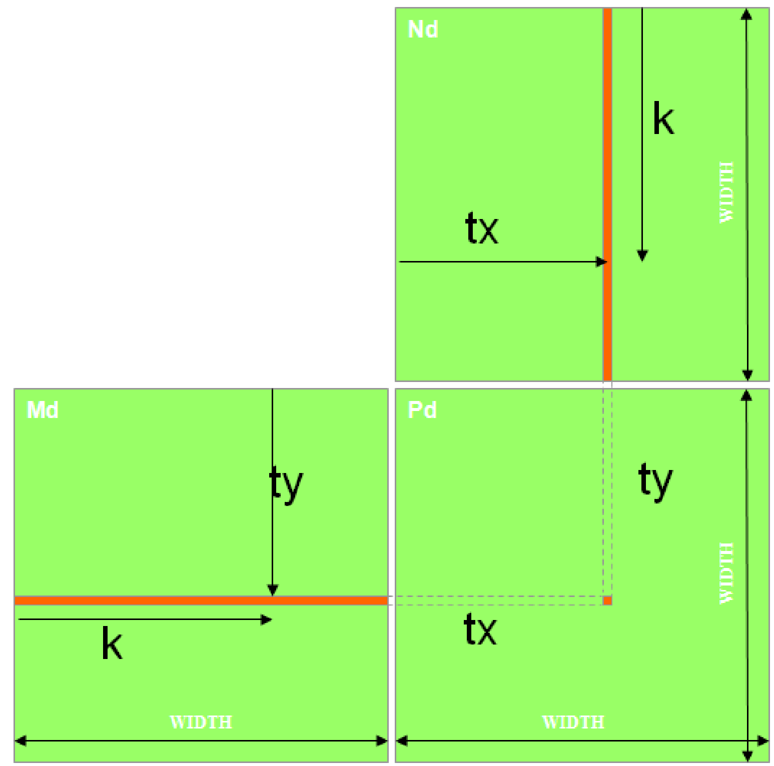
Step 4: Kernel Function




```
// Matrix multiplication kernel – per thread code
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;

   for (int k = 0; k < Width; ++k) {
       float Melement = Md[threadIdx.y*Width+k];
       float Nelement = Nd[k*Width+threadIdx.x];
       Pvalue += Melement * Nelement;
   }

  Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



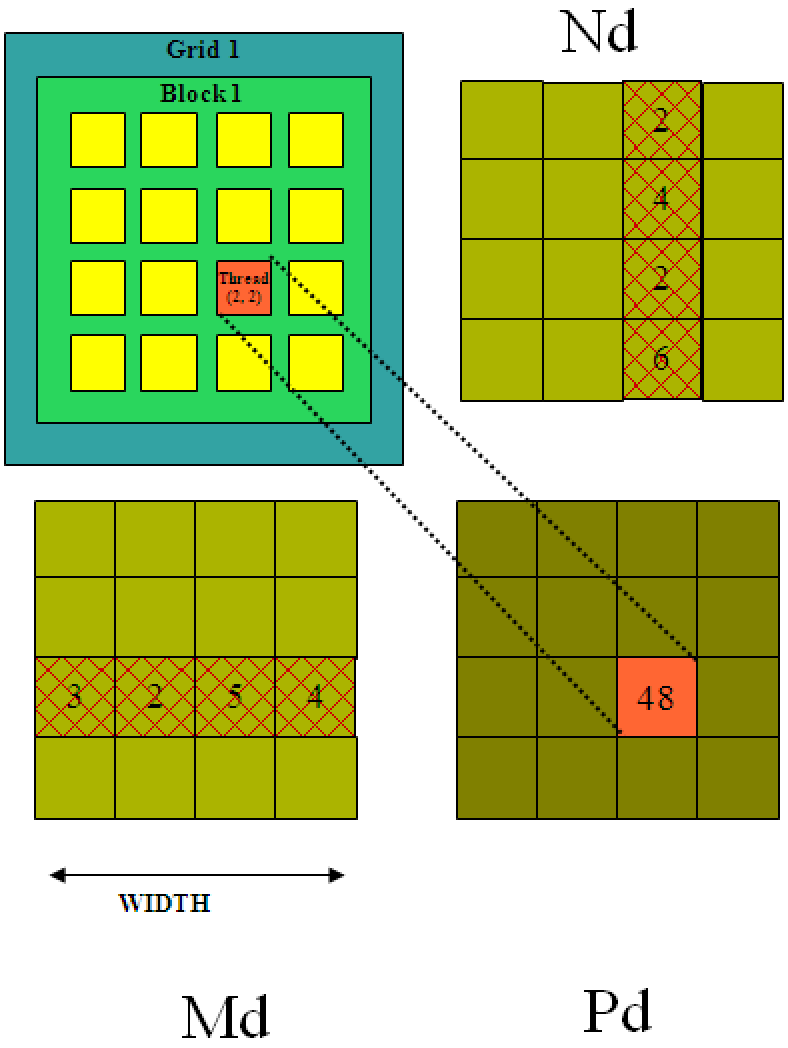
Step 5: Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
    dim3 dimGrid(1, 1);
    dim3 dimBlock(Width, Width);


  // Launch the device computation threads!
  MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

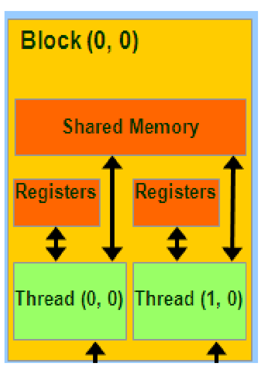
Only One Thread Block Used

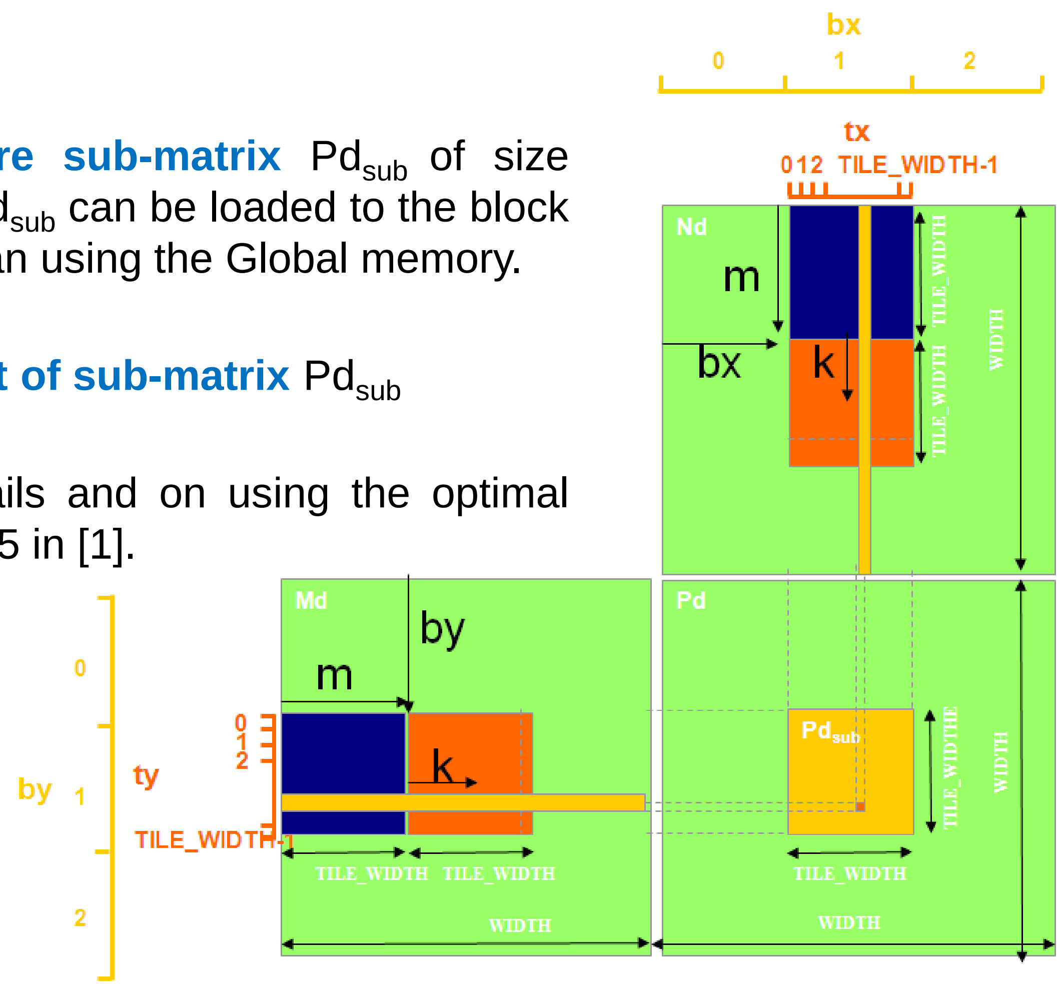
- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1
- **Size of matrix limited** by the **number of threads allowed in a thread block.**
- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 4*346.5 = 1386 GB/s required to achieve peak FLOP rating
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code runs at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



Using Tiles and Multiple Blocks

- **Break up** the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd.
- Each **block** computes **one square sub-matrix** Pd_{sub} of size TILE_WIDTH and so a Nd_{sub} and Md_{sub} can be loaded to the block shared memory for faster access than using the Global memory.
- Each **thread** computes **one element of sub-matrix** Pd_{sub}
- For more details on the code details and on using the optimal block size please read chapters 4 & 5 in [1].



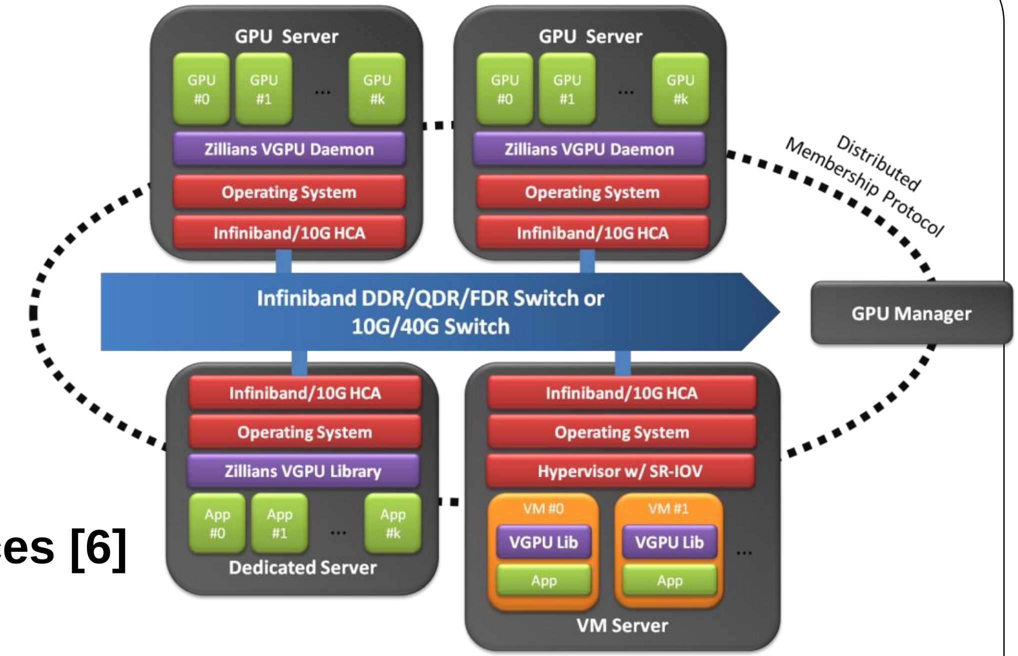


Current trends in the GPGPU research

(Cloud computing, DBMS and Data Mining, Networks and Security)

Cloud Computing

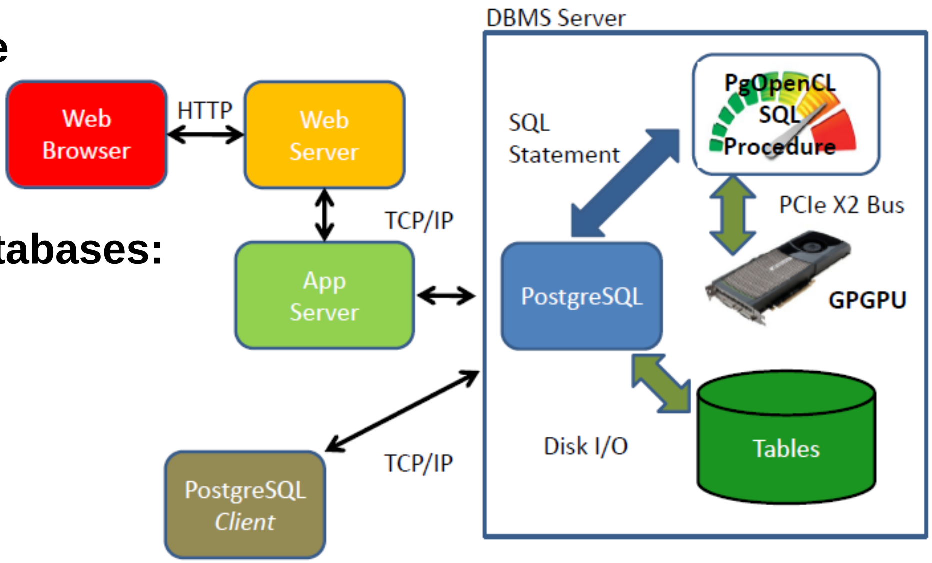
- **GPGPU Virtualization:**
  Sharing GPU power between users in the cloud with a pay-as-you-go strategy.



  - **Amazon EC2 Cluster GPU instances [6]**
  - **Zillians GPU Virtualization [7]**
  - **GPU virtualization on VMware's hosted I/O architecture [8]**
  - **GPU Cluster for High Performance Computing [9]**
  - **Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework [10]**

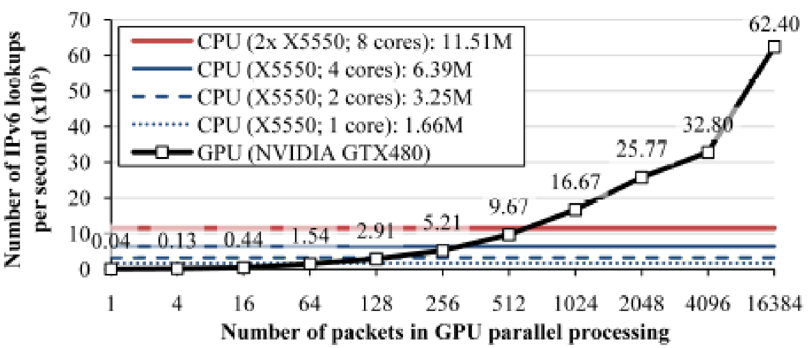
DBMS and Data Mining

- **Mars: Accelerating MapReduce with Graphics Processors [11]**
- **A New PostgreSQL Procedural Language Unlocking the Power of the GPU [12]**
- **Hardware acceleration in commercial databases: a case study of spatial operations [13]**
- **GPUQP: Query Co-Processing Using Graphics Processors [14]**



Network and Security

- **GPU packet classification using OpenCL: a consideration of viable classification methods [15]**
- **Parallel packet classification using GPU co-processors [16]**
- **Efficient GPGPU-Based Parallel Packet Classification. [17]**
- **Acceleration of packet filtering using gpgpu [18]**
- **Research into GPU accelerated pattern matching for applications in computer security [19]**
- **Hermes: an integrated CPU/GPU microarchitecture for IP routing [20]**
- **PacketShader: a GPU-accelerated software router [21]**



Quiz

- Now after knowing everything about the massive processing power of GPUs.
  - Mention a problem that using GPUs would help.

CUDA

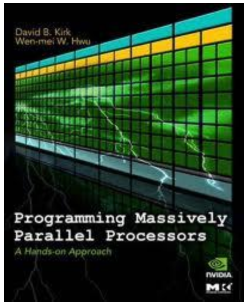
- Toolkit
  http://developer.nvidia.com/cuda-downloads

- List of CUDA enabled GPUs : http://developer.nvidia.com/cuda-gpus

- If you find this topic interesting to you, I recommend reading the book in reference 1 and checking the course in http://courses.engr.illinois.edu/ece498/al/

References

1) David B. Kirk and Wen-mei W. Hwu, “Programming Massively Parallel Processors: A Hands-on Approach”, Morgan Kaufmann Publishers, 2010. ISBN: 978-0-12-381472-2



2) http://en.wikipedia.org/wiki/Intel_Core
3) http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf
4) http://perspectives.mvdirona.com/2009/03/15/HeterogeneousComputingUsingGPGPUsNVidiaGT200.aspx
5) http://www.nvidia.com/object/cuda_home_new.html
6) http://aws.amazon.com/hpc-applications/
7) http://www.zillians.com/vgpu/how_it_works
8) Micah Dowty and Jeremy Sugerman. 2009. GPU virtualization on VMware's hosted I/O architecture. *SIGOPS Oper. Syst. Rev.* 43, 3 (July 2009), 73-82.
9) Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. 2004. GPU Cluster for High Performance Computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*(SC '04).
10) Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. 2011. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In*Proceedings of the 20th international symposium on High performance distributed computing*(HPDC '11).
11) Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K. Govindaraju. 2011. Mars: Accelerating MapReduce with Graphics Processors. *IEEE Trans. Parallel Distrib. Syst.* 22, 4 (April 2011), 608-620.
12) http://wiki.postgresql.org/images/6/65/Pgopencl.pdf
13) Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. 2004. Hardware acceleration in commercial databases: a case study of spatial operations. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30* (VLDB '04), 1021-1032.
14) http://www.cse.ust.hk/gpuqp/
15) Alastair Nottingham and Barry Irwin. 2009. GPU packet classification using OpenCL: a consideration of viable classification methods. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists* (SAICSIT '09).
16) Alastair Nottingham and Barry Irwin. 2010. Parallel packet classification using GPU co-processors. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists* (SAICSIT '10).
17) Che-Lun Hung, Hsiao-Hsi Wang, Shih-Wei Guo, Yaw-Ling Lin, and Kuan-Ching Li. 2011. Efficient GPGPU-Based Parallel Packet Classification. In *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*(TRUSTCOM '11).
18) Manoj Singh Gaur, Vijay Laxmi, Lakshminarayanan V., Kamal Cahndra, and Mark Zwolinski. 2011. Acceleration of packet filtering using gpgpu. In *Proceedings of the 4th international conference on Security of information and networks* (SIN '11).
19) http://www.csse.canterbury.ac.nz/research/reports/HonsReps/2009/hons_0905.pdf
20) Yuhao Zhu, Yangdong Deng, and Yubei Chen. 2011. Hermes: an integrated CPU/GPU microarchitecture for IP routing. In *Proceedings of the 48th Design Automation Conference* (DAC '11).
21) Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference* (SIGCOMM '10).

Questions