GPU Programming

0) INTRODUCTION

In modern processors, quad-core processors are very common, with 6-8 cores for each CPU and fast network adapters even in motherboards. Also, **MULTITHREADING** is growing (more HW threads in architecture are supported, managed with HW accelerator modules).

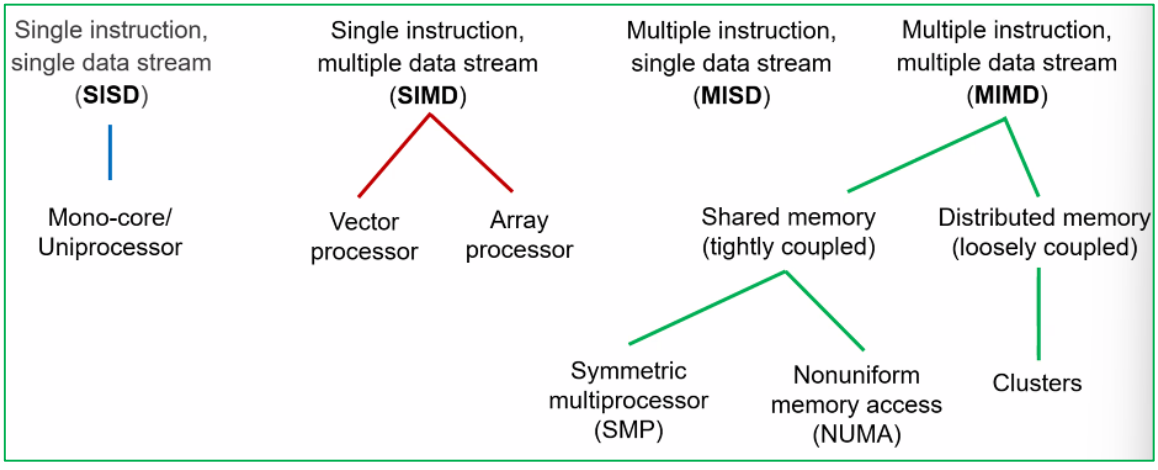
PARALLELISM is executing multiple tasks at the same time

- **ILP** (**Instruction Level Parallelism**) has features as:
  - Pipeline parallelism
  - Superscalar
  - Out-of-order execution
  - Branch prediction
  - Speculative execution
- **Task parallelism** → distribution of tasks concurrently across different processors, that are executed through *processes*/*threads* (GPU mostly works with threads)
- **Data parallelism** → concurrent execution of a task on different data

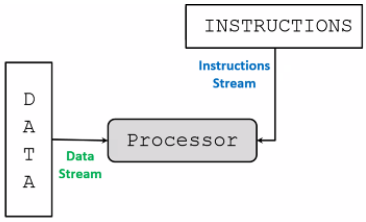
We can also do parallelism through **distributed computing** (multiple machines working together, network-based communication), with nodes and network interface cards (more GPUs used in system).

COMPUTER ARCHITECTURES CLASSIFICATION

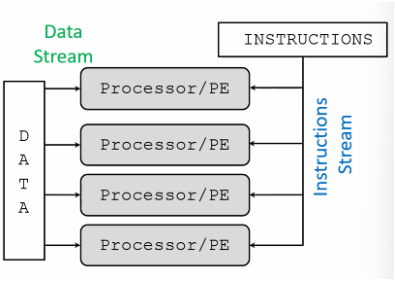
Flynn's Taxonomy is a classification system for computer architectures, based on the handling of instructions and data:



- **SISD** → a single processor executes a single instruction stream to operate on data stored in a single memory (mono-core and uniprocessors) [so the stream is executed serially because 1 CPU] [è l'approccio usato finora nella programmazione senza thread]
  Performance: $\boldsymbol{MIPS_{rate} = f \cdot IPC}$ (MIPS = millions of instructions per cycle; IPC = instruction per cycle)



- **SIMD** → each PE (processing element) has an associated data memory, so that each instruction operates different sets of data by the different processors [used in **VPU**s (vector processes that act on array of similar data and are several times faster than when executing in scalar mode), domain specific **accelerators**, **some GPU**s] [**processor arrays** use more processing units to execute in parallel the same instruction on different data]



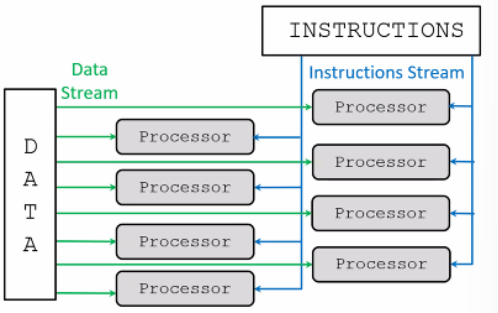
⚠ We have to understand the specific architecture (and so its library) to write programs for them [here example of Intel SSE]:

```c
#include <stdio.h>
#include <xmmintrin.h> // SSE header
int main() {
    // Arrays must be 16-byte aligned for SSE
    float a[4] __attribute__((aligned(16))) = {1.0, 2.0, 3.0, 4.0};
    float b[4] __attribute__((aligned(16))) = {5.0, 6.0, 7.0, 8.0};
    float result[4] __attribute__((aligned(16)));
    // Load arrays into SIMD registers
    __m128 vecA = _mm_load_ps(a);
    __m128 vecB = _mm_load_ps(b);
    // Perform SIMD addition
    __m128 vecResult = _mm_add_ps(vecA, vecB);
    _mm_store_ps(result, vecResult); // Store the result back to memory
    for (int i = 0; i < 4; i++) {    // Print the result
        printf("result[%d] = %.1f\n", i, result[i]);
    }
    return 0;
}
```

```
g++ -std=c++11 -O3 -msse Vector_Add.cpp -o vector
```

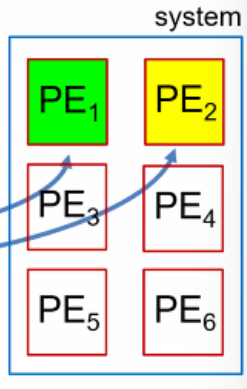
- **MISD** → only theoretical model (not used by us) where a sequence of data is transmitted to a set of processors, each executing a different instruction sequence of the same stream

- **MIMD** → a set of processors/PEs simultaneously execute different instruction sequences on different data sets (**GPUs**, **SMPs**, **clusters**, **NUMA**). The processor's instructions and data are connected because they are all sub-parts of the same overall task



- **Coding example of MIMD**:

```c
// Operative function to add or mul a constant to each element
void Operative_function(float* data, float value, int sel) {
    for (int j= 0; j < 100 ; j++) {
       if (sel ==1)
          data[j] = data[j] + value;
       else:
          data[j] = data[j] * value;
    }
}

int main() {
    …
    // Execute the functions with diff input arrays and configurations
    Operative_function(queue1)(array_data1, 5.0f, 1);
    Operative_function(queue2)(array_data2, 10.0f, 0);
    …
    return 0;
}
```



MIMD systems are mainly (sometimes mixed):

- **SM** (Shared memory) → multiple CPUs share the same address space [only 1 memory]; it can be used **OpenMP** (standard)
- **DM** (Distributed memory) → each CPU has its own memory; it can be used a "message passing" paradigm (Parallel Virtual Machine [PVM] or Message Passing Interface [**MPI**; standard])

[QUI LA STORIA DELLE GPU se richiesto; qui sotto esempio di calcolo dei pixel]

In the early 2000s, there was the rise of **programmable shaders** (software to help the GPU) [**vertex shaders** (control each vertex of a 3D polygon) and **pixel shaders** (small programs that run on a GPU to determine colors, lighting, shading, textures, materials…)]. In mid-2000s GPUs started to be used also in non-graphics sectors (math, physics, crypto…); shaders become threads.
GPU acceleration is used to handle the computational load of training and executing deep neural networks (with addition of specialized in-chip accelerators as tensor/matrix cores and ray tracing HW).

1) DISTRIBUTED COMPUTING

Distributed Computing is "a system in which the failure of a computer you didn't even know existed can render your own computer unusable". Task is divided across multiple computers (*nodes* or machines) that can work together to solve it [share the workload] (because some tasks are too complex to solve by a single device in a decent time)
[**Scalability** → systems can grow by adding more machines (easier to handle larger tasks); **Resource sharing** → machines can share data, storage and processing power]

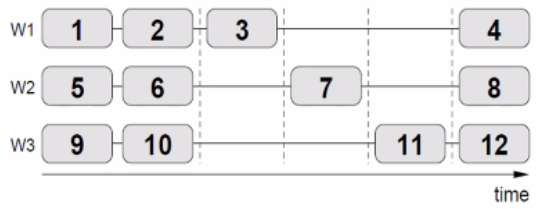
Parallel Processing is execution of multiple tasks (or parts of task) by dividing them across multiple processors/cores or PE (processing elements) in the same computer system. Principal concepts are **task division** and **shared memory**. Goal → finish computation faster than if they were done sequentially

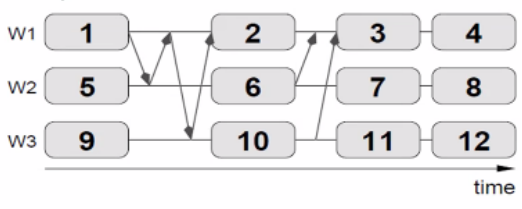
⚠ **Not all the tasks in a given work can be parallelized** because of:
- startup overhead
- algorithmic limitations (es. mutual dependencies)
- bottlenecks (es. shared resources)
- communication (synchronism)

So, the total amount of work is composed by:

$$\textbf{Total amount of work} = \boldsymbol{Tw} = \boldsymbol{s} + \boldsymbol{p} \qquad \text{con} \begin{cases} \boldsymbol{s} = serial\ part \\ \boldsymbol{p} = parallel\ part \end{cases}$$

And the total amount of time required for work is $\boldsymbol{T_w} = \boldsymbol{t_s} + \boldsymbol{t_p}/\boldsymbol{N_{PEs}}$ (N_{PEs} is the n° of workers):





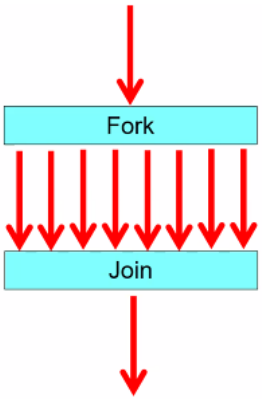
Speedup (*Amdahl's law*) is gain obtained in parallel programming (> parallelizable code → > speedup):

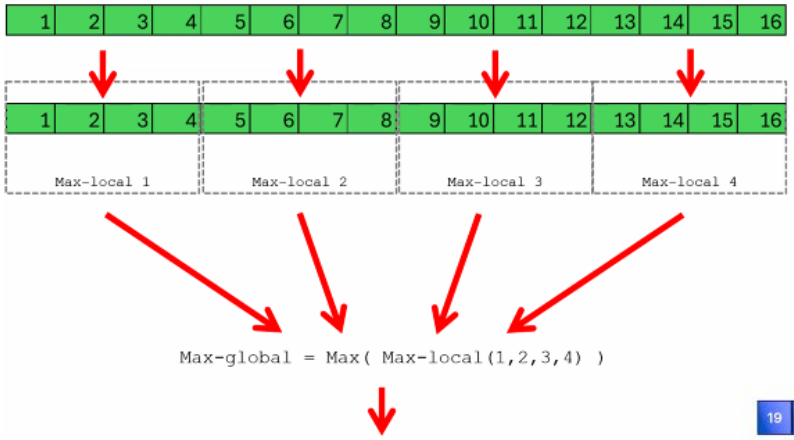
$$\boldsymbol{Speedup} = \frac{\boldsymbol{1}}{(\boldsymbol{1}-\boldsymbol{p}) + \frac{\boldsymbol{p}}{\boldsymbol{n}}} \qquad con \begin{cases} \boldsymbol{p} = \dfrac{parallelizable\ code}{total\ execution\ time} \\ \boldsymbol{n} = n°\ of\ processors\ the\ code\ can\ use \end{cases}$$

Example: if there are **4** processors and only **10%** of the code is parallelizable
Speedup = 1/(0.9+(0.1/4))=1.081, that is only **8%** with **4** procs!!

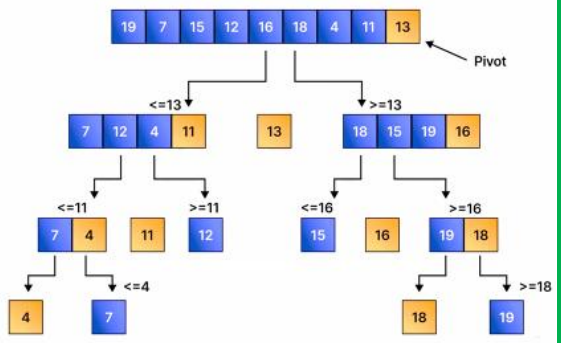
PARALLEL ALGORITHMS:

- **FORK-JOIN** = a process is divided in parallel threads working concurrently (FORK); finally, all the contributions are put together in the JOIN phase. Exploits shared-memory parallelism. **OpenMP** supports this parallelism strategy
  Example (max element of array):

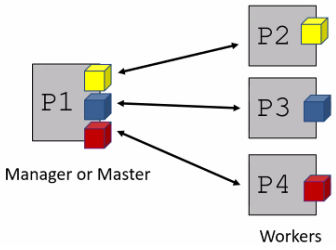




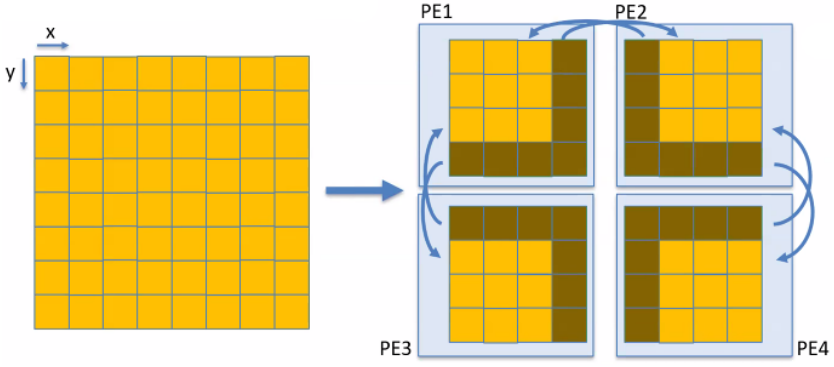
- **DIVIDE & CONQUER** = break the problem in smaller subproblems having similar algorithmic properties to the original one, but not join at the end. Recursion can be used in this kind of programming strategy
  Example (Quick-sort algorithm):



- **MANAGER-WORKER** (or MASTER-SLAVE) = manager process distributes the tasks and collects the results from worker processes, which elaborate and return the result to manager; **PthTreads** and **MPI** support
  - **Embarrassingly Parallel** = subclass of Manager-Worker, with lot of parallelism with essentially no inter-task communication; require reduction operation at the end to gather results (Example = operazioni matematiche slegate tra loro che posso eseguire quante volte voglio)



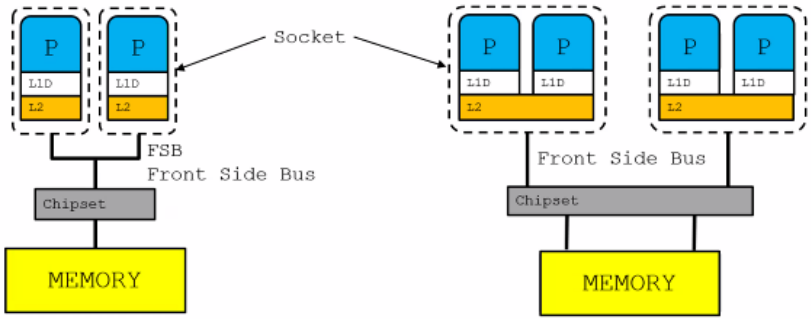
- **HALO EXCHANGE** = in distributed computing it's the process of communicating the boundary data between neighboring subdomains, so each operation can continue correctly (usually involves sending boundary values to neighbors and receiving their boundaries to fill ghost cells)
  Example (split a grid/array across multiple processors /threads):



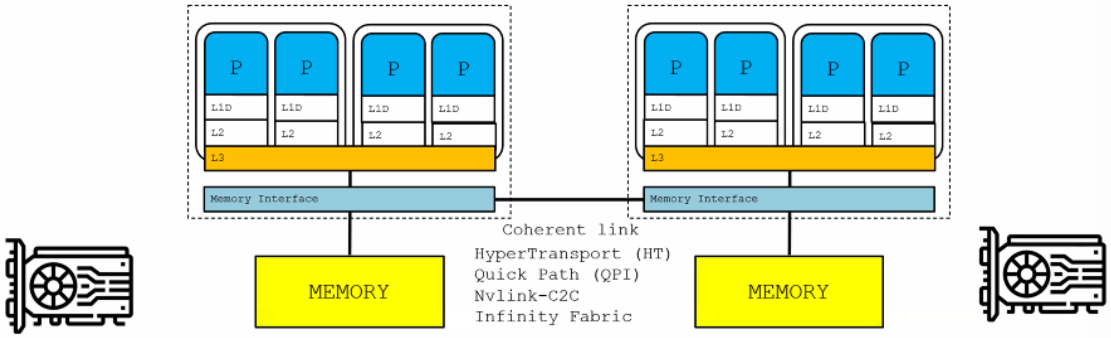
- **Permutation algorithms** = procedure to generate arrangements of elements efficiently (**permutazioni** → {A, B, C} → ABC, ACB, BAC, BCA, CAB, CBA) [es. used in cryptography]

SHARED-MEMORY in Distributed Computing:

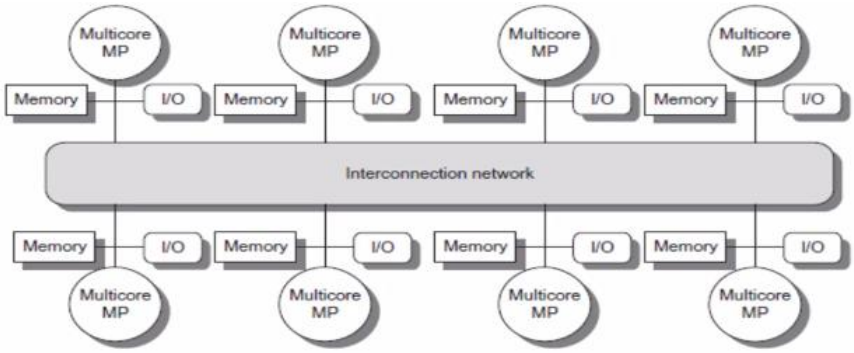
- **SMC** (Shared-Memory Computer) = a small n° of CPUs work on a **common shared physical address space** (common memory is shared by all the processors with uniform memory latency)
- **UMA** (Uniform Memory Access) = SMC that exhibits a flat memory model; **latency and bandwidth** are the same for all processors and memory locations. **SMP** (Symmetric Multiprocessing)



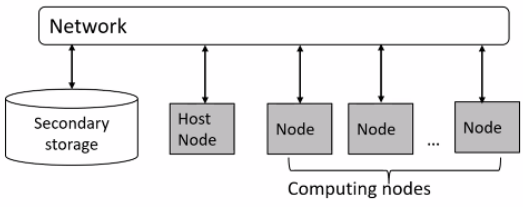
- **ccNUMA** (Cache-Coherent Non-uniform Memory Access) = SMC where the memory is **physically distributed** (HW) but **logically shared** (SW). A **LD** (Locality Domain) is composed by some processing elements (PE) and a locally connected memory; any processor can access to any other processor's memory in the LD. Base of modern GPUs



- **DMP** (Distributed-Memory Processor) = composed by **different multicores with memory and I/O attached to them**. It's a **NUMA**; processors are connected via direct and non-direct *interconnection networks* (usually these systems are called **MPP** [Massively Parallel Processors])



- **HYBRID SYSTEMS** = large parallel computers are mix of shared memory and distributed systems
- **COMMODITY CLUSTERS** = group of standalone computers able to work independently, connected through an integration network developed independently. Mass storage devices are **COTS** (Components Off The Shelf) installed within the cluster nodes (or externally); they use **MPI** and/or **OpenMP**



LIBRARIES for PARALLEL PROGRAMMING (Threads, MPI and OpenMP)

THREAD = independent stream of instructions that can be scheduled to run by the OS. It's a lightweight process that executes independently but shares the parent process's memory. It has its own execution context and it's the smallest sequence of programmed instructions that can be managed independently by a scheduler.

Process has its own: PID, GID, UID, environment, working directory, instructions, registers, stack, heap…
Thread has its own Stack pointer, Registers, Scheduling properties, Signals, Thread specific data

Since threads share resources within the same process:

- If 1 thread changes 1 or more shared system resources, all the other threads will see the change
- If 2 threads have 2 pointers containing the same address, they point to the same data
- **Explicit synchronization** by the programmer is required since it's possible to read and write at the same time the same memory locations

General rules of **POSIX Threads** (`pthread` in C++):

- thread function signature → `void* function(void* arg)` [it must return void* and accept 1 void*]
- creating threads:

```
pthread_create(&thread_id, nullptr, threadFunc, arg);
```

- waiting for threads (joining):

```
pthread_join(thread_id, nullptr);
pthread_detach(thread_id);
```

- Synchronization:

```
pthread_mutex_lock(&mutex);
// critical section
pthread_mutex_unlock(&mutex);
```

- avoid passing local variables directly (use **heap-allocated memory**):

```
// Allocate a single integer on the heap
int* ptr = new int;  // ptr points to an int on the heap
*ptr = 42;           // store value in heap memory
delete ptr;          // Deallocate memory to avoid memory leak
```

- clean up resources:

```
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&cond);
```

- be careful with **exceptions** (use `try-catch` in thread functions)
- **minimize share data** (prefer passing copies of data than sharing them)

MPI = message passing library for large distributed systems (portable, efficient, flexible, standard); not directly used for GPUs, but for distributed memory systems (and in some cases for shared memory systems) [**OpenMPI** is the best version] [it's strong with graphs]

How to use it?

```
#include "mpi.h"
// Inizialize MPI environment
// Message passing calls
// End of MPI environment
```

- `MPI_Init(&argc, &argv)` initializes MPI
- `MPI_Finalize()` close MPI
- `MPI_Comm_Size(communicator, &size)` gets the number of processes within the communicator
- `MPI_Comm_rank(communicator, &rank)` gets the rank of the calling process within the communicator
- `MPI_Abort(communicator, errorcode)` terminates all processes, often regardless of specified communicator
- `MPI_Get_processor_name(&name, &resultlenght)` gets name and lenght of processor
- `MPI_Inizialized(&flag)` verifies if MPI_Init has been called
- `MPI_Wtime()` returns the time of the calling processor
- `MPI_Wtick()` gives resolution in seconds of MPI_Wtime

Point to Point Communication Routines (message passing between only 2 MPI tasks [1 send, 1 receive]) thanks to MPI that guarantees message order. MPI doesn't include "*operation starvation*".

OpenMP (Open Multi Processing) = more powerful in local systems. It's not a new computer language, but it works in conjuction with other languages. It has a specific API. It's used to **distribute and decompose the work across multiple processors, using threads** to deploy work. OpenMP implements **Fork-Join** model. With very few lines, parallelism done. Uses the compiler directories (`#pragma`) Do attention:

```c
#include <stdio.h>
#include <omp.h>
main()
{
   int nthreads, tid;
   printf("Hello parallel world from threads:\n");
   /* set the number of threads
    (maybe greater than number of core/processors) */
   omp_set_num_threads(8);
   // fork
   #pragma omp parallel private(tid)
   {
     printf("%d\n", omp_get_thread_num());
   }
   // implicit join given by the "}"
   printf("Back to the serial world.\n");
}
```

```
!$omp parallel
   do i = 1,10
     print *, 'Hello world', i
   enddo
!$omp end parallel
```

given 4 threads, it prints 4*10= 40 messages

Why?

```
!$omp parallel do
   do i = 1,10
     print *, 'Hello world', i
   enddo
```

given 4 threads, it prints 10 messages

⚠ Statistical Profilers = collecting data whenever a sampling interval expires (fast, but not exact); Instrumenting profilers = insert code in the program to measure (exact, but slow; can alter synchronization)

##

⚠ LAB01 – conclusioni

1. Caratteristiche principali di Threads, pthreads, OpenMP e MPI

- **Threads (std::thread in C++):**
  Consentono l'esecuzione concorrente di più flussi di istruzioni all'interno dello stesso processo. Condividono la memoria e le risorse del processo principale. Sono facili da usare in C++ moderno.
- **pthreads (POSIX threads):**
  API standard per la gestione dei thread su sistemi Unix/Linux. Più "bassa" e manuale rispetto a std::thread. Richiede gestione esplicita di creazione, sincronizzazione e terminazione dei thread.
- **OpenMP:**
  Libreria per il parallelismo a livello di thread tramite direttive nel codice (#pragma omp). Permette di parallelizzare facilmente cicli e sezioni di codice. Gestisce automaticamente la creazione e sincronizzazione dei thread.
- **MPI (Message Passing Interface):**
  Standard per il parallelismo distribuito: esegue più processi, ognuno con la propria memoria. I processi comunicano tramite messaggi (send/receive). Usato per calcolo su cluster e supercomputer.

2. Differenza tra parallel computing e distributed computing

- **Parallel computing:**
  Più thread/processi lavorano in parallelo sulla stessa macchina, condividendo la memoria (es. OpenMP, thread, pthreads).
- **Distributed computing:**
  Più processi (spesso su macchine diverse) lavorano insieme, ognuno con la propria memoria, comunicando tramite rete (es. MPI).

3. Si possono combinare? Esempio pratico

Sì, è possibile combinare parallel computing e distributed computing.
Un esempio tipico è l'approccio ibrido MPI+OpenMP:

- **Esempio:**
  In un cluster HPC, ogni nodo esegue un processo MPI. All'interno di ogni nodo, il processo MPI avvia più thread OpenMP per sfruttare tutti i core disponibili. Così si sfrutta sia il parallelismo distribuito (tra nodi) sia quello condiviso (all'interno del nodo).
- **Altri esempi:**
  - Simulazioni scientifiche su supercomputer.
  - Analisi di big data su cluster (es. Apache Spark usa processi distribuiti e thread interni).

##

3) GPU Organization and Concurrency

A **CPU** (Central Processing Unit) is used by applications for primary calculations; most programming models focus on CPUs, because they have general purpose capabilities.

A **GPU** (Graphics Processing Unit) is designed for parallelizable problems (in the beginning only graphics, now more capable of general computations [edge computing, *high performance* computing]). They are very fast and powerful, but consumes a lot of electrical power. It's a device equipped with microprocessors based on **SIMD/MIMD** paradigms and a private memory with high bandwidth (it's a specialized parallel processor). A GPU exploits its **implicit parallelism** and **programming flexibility** to boost performance of complex and data-intensive workloads (**parallel intensive computation**).

Rendering process requires a set of transformations based on linear algebra operations, tensor operations and filters. The **same set of operations** are applied on each data point of the scene; **each operation is independent with respect of data** (data not related, not wait the result of another operation). All operations are performed in parallel using a lot of threads.

A **THREAD** in GPU is an independent flow of execution of a program which share the same memory resources of the main process. Threads inside a process can **exchange data** (shared memory of the program), **use local memory** (not shared with other threads) and **synchronize** (with other threads).

SPMD (Single Program Multiple Data) **Parallelism** = if the set of transformations can be applied independently on each point, output is independent on the order of point computation; so if transformations are independent, we can speed up the elaboration using parallel work:

apply the same transformation (SP) to each point [data] (MD)

```
// typical loop over each point with the same set of operation
for each point in collection_of_points:
    output_data = trasformations_on_point(point, input_data)
```

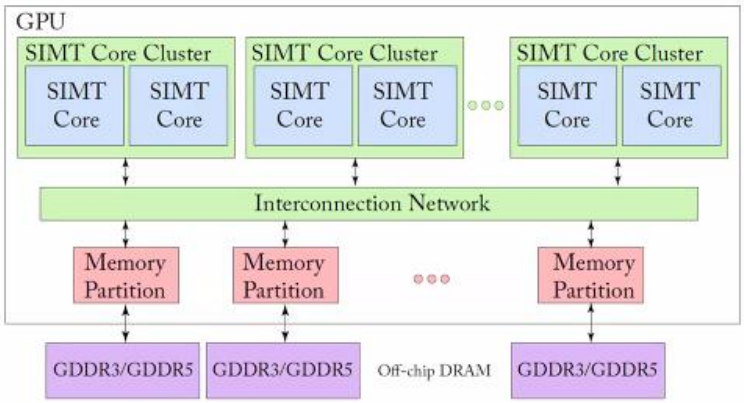
Abbiamo già visto Amdahl's Law (che dice che posso ridurre i tempi computazionali solo delle parti parallel e non serial, che rimangono uguali anche se aggiungo core che lavorano in parallelo), ma poi arrivò **Gustafson's Law**: the serial fraction of a task doesn't theoretically limit parallel speedup, if the problem or workload scales in its parallel component: task's speedup increases when the problem size scales along with the number of processors, rather than remaining fixed

$$\boldsymbol{Speedup = P + s(P - 1)}$$

In **CPU**, when there are more threads than available cores, the OS can make a context switch (running thread is freezed, a new thread take the HW resources). But in **GPU**, there are THOUSAND of threads: each GPU thread is *light weighted*; GPU have thousands of available registers, so **no penalty in case of context switch** (> threads in flight, < delays).

GPU ORGANIZATION

CPU hw minimize latency using large caches and complex control logic; **GPU hw** hides memory and instruction latencies with computing operations.



So a **GPU architecture** is composed by an amount of clusters (**SM** [Stream Multiprocessors] **cores**) and a main global memory; each SM has many cores (**PE**s: FP32, INT32, FP64, SFU [Special Functional Unit], LS [Load/Store], Tensor Cores [a * b + d]), lots of registers, instruction scheduler dispatchers and a share memory with very fast access to data.

Warp = group of consecutive threads that execute instructions in lockstep (run the same instruction at the same time on different data) [is the smallest scheduling unit in NVIDIA GPUs (32 threads) → in fact in NVIDIA GPUs the *SM warp scheduler* schedules threads in groups of 32 (warps); if using 2 warp schedulers per SM, allows 2 warps to be issued and executed concurrently if hw resources available].

⚠ To connect more GPUs together, **NVLink** technology
⚠ Separating FP32 and INT32 cores, allowing simultaneous execution of FP32 and INT32 operations at full throughput

GPGPU (**General Purpose** GPU) **Programming** relates to use of GPU to solve problems other than graphics. GPU is seen as an *auxiliary coprocessor* equipped with thousands of cores and a high bandwidth memory (except from NPU, where GPU and CPU are not separated). **Serial parts** of a program run **on CPU** (**HOST**), **parallel parts on GPU** (**DEVICE**); data must be moved from HOST to DEVICE memory to be processed on the GPU (usually this data movement is the **bottleneck** of many GPU porting activities) [in fact if the problem is too easy, data transfer can take more than the GPU computation]; when data is processed, it's transferred back to HOST.

A function running on GPU is called "**kernel**": when a kernel is launched, thousands of threads will execute its code; programmer chooses the number of threads to launch.

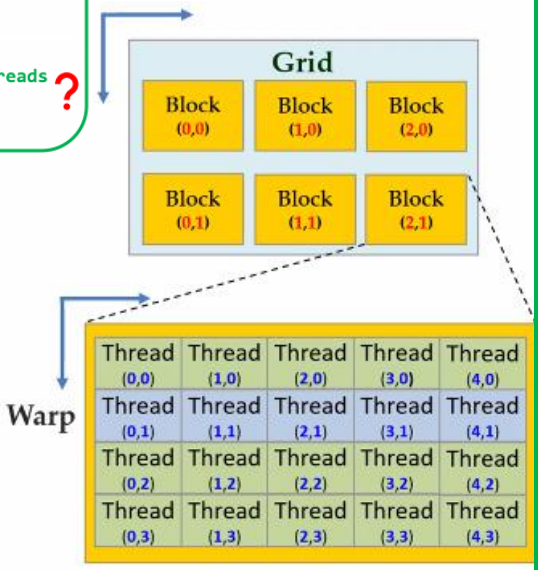
Example: vector addition

One thread

```
void vecAddCPU (int N, const float *A,
                const float *B, float *C)
{
    for ( int i = 0; i < N; i++ )
        c[i] = a[i] + b[i];
}
...
// call vecAddCPU on N elements
vecAddCPU ( N, a, b, c );
```

N threads

```
void vecAddGPU (int N, const float *A,
    const float *B, float *C)
{
    int i = .... // use unique thread index ?
     if ( i < N) c[i] = a[i] + b[i];
}
...
// launch kernel with 1 block of N threads ?
vecAddGPU<<<1, N>>>( N, a, b, c );
```



GPU Thread Hierarchy → to compute *N* elements (1 per thread) in parallel on a GPU, min *N* concurrent threads must be created; GPU threads are grouped in thread blocks (**Warp**); threads belonging to the same block can cooperate together exchanging data through a shared memory area.

⚠ The organization of these Threads-Warps-Blocks-Grids are multidimensional (not 2D, but **3D**).

⚠ GPU can execute blocks in any order: this flexibility is important to have the min time possible!

Latency = clock cycles needed to compute an operation; we can't avoid it, but we can HIDE LATENCIES: **the code can be organized to provide to the scheduler a sufficient number of independent operations so that, the more the warp are available, the more context-switch is done** [come quando facevamo noi i compilatori umani ad ASE con il rescheduling, ma su GPU con migliaia di threads].

⚠ **TLP** (Target-Level Parallelism): try to provide as much threads per SM as possible, so the scheduler find a warp ready to execute, while the othere are busy

⚠ **ILP** (Instruction-Level Parallelism): try to differentiate the same type of operations in different SM (so each can do 1 of the required operations, and later the results will be combined → es. se devo fare 3 FP operations e 3 INT operations, non faccio 2 FP in un SM solo, ma spacco 1FP in ognuno dei 3 SM così ci metterò meno e l'unità FP non sarà in wait per la 2^ operation)

How to program for a GPU?

- **identify compiler framework** (NVCC/HIPCC)
- **describe program**
  - configure device + initialize GPUs memory
  - define parallelism according to the application (# of concurrent threads)
  - execute program on GPU
  - collect/retrieve the results to the host
- **compile and link**
- **execute**

EXAMPLE of SAXPY in CUDA:

Saxpy.cu

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cuda_runtime.h>

// Kernel function (GPU) to perform SAXPY operation:
__global__ void saxpy(float a, float *X, float *Y, int n)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < n) {
        Y[idx] = a * X[idx] + Y[idx];
    }
}
```

Configuration of the thread redundancy

⚠ There's fine *granularity* index for threadIdx (Thread Index) because we can have **`threadIdx.x`**, **`threadIdx.y`**, **`threadIdx.z`**; for Block Index is similar, but only **`blockIdx.x`** and **`blockIdx.y`** (and also for **`blockDim.x`** and **`blockDim.y`**) [a seconda di quello che ci serve o della complessità, possiamo usarli]

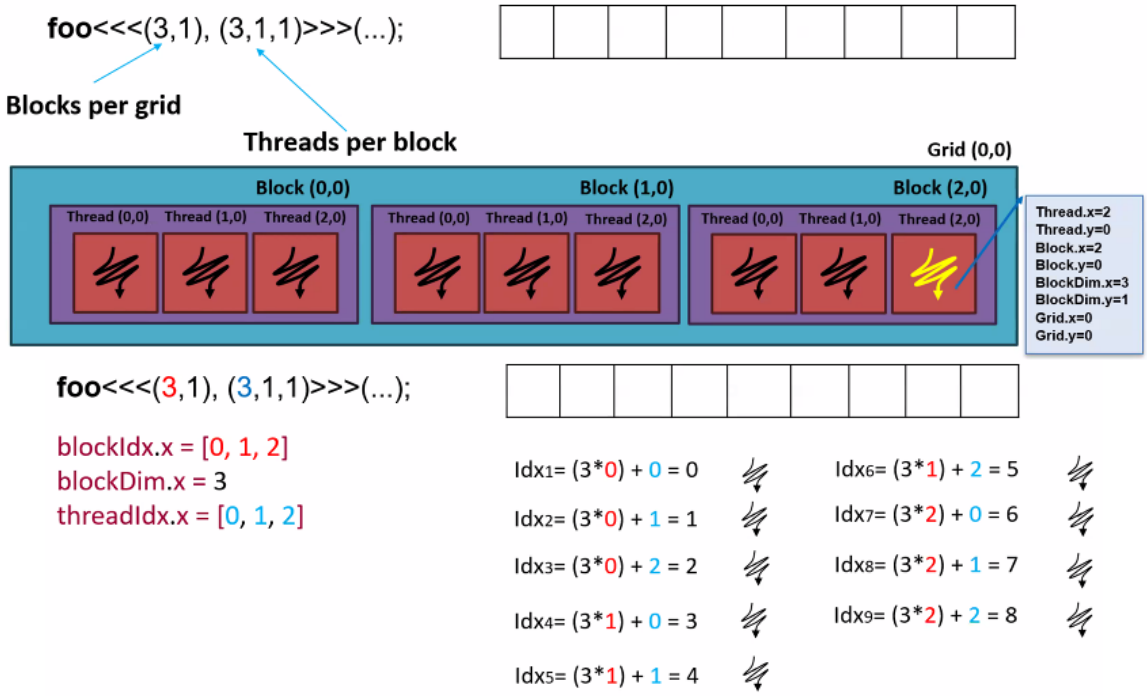
And then the main of the SAXPY above:

Saxpy.cu

```
int main() {
    const int N = 10000;                        // Number of elements
    const int size = N * sizeof(float);
    const float a = 2.5;                        // Scalar value for SAXPY
    std::vector<float> h_X = readVectorFromFile("vectorX.txt", N);
    std::vector<float> h_Y = readVectorFromFile("vectorY.txt", N);

    float *d_X, *d_Y;

    // Allocate device memory
    cudaMalloc((void **)&d_X, size);
    cudaMalloc((void **)&d_Y, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_X, h_X.data(), size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_Y, h_Y.data(), size, cudaMemcpyHostToDevice);

 // Launch the kernel on the GPU
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    saxpy<<<blocksPerGrid, threadsPerBlock>>>(a, d_X, d_Y, N);

    // Copy result vector from device memory to host memory
    cudaMemcpy(h_Y.data(), d_Y, size, cudaMemcpyDeviceToHost);

    // Write the result vector to a file
    writeVectorToFile("result.txt", h_Y);

    // Free device memory
    cudaFree(d_X);
    cudaFree(d_Y);

    return 0;
}
```

Imagine thread organization as **unique thread indexes**; handling parallel configuration parameters:

- 1-dimension arrays → **`idx = blockDim.x * blockIdx.x + threadIdx.x`**



⚠ Index limit are associated to GPU architecture (so we need to know it to determine the limits)!

EXAMPLE (C and CUDA – calcolo del prodotto matriciale):

In C:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{m} A_{ik} * B_{kj}$$

```
void multiplyMatrix(int A[][n1], int B[][n1])
{
    int C[m1][m1];

    for (int i = 0; i < n1; i++) {
        for (int j = 0; j < n1; j++) {
            C[i][j] = 0;

            for (int k = 0; k < m1; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
            printf("%d\t", result[i][j]);

        }
        printf("\n");
    }
}
```

In CUDA:

$$\sum_{k=1}^{n} A_{ik} * B_{kj}$$

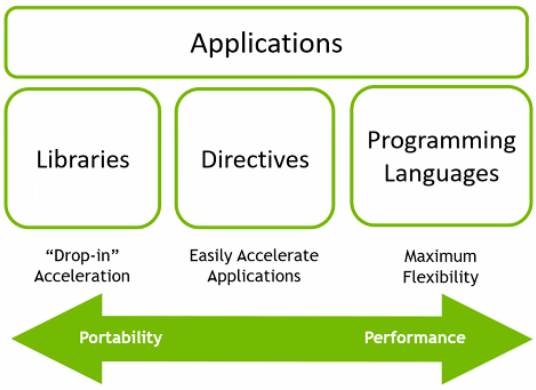
i = blockDim.y * blockIdx.y + threadIdx.y;
j = blockDim.x * blockIdx.x + threadIdx.x;

```
__global__ void kernelMatrixMult(float* C, float* A,
             float* B, size_t N,
              size_t M){
    size_t i = blockIdx.y * blockDim.y + threadIdx.y;
    size_t j = blockIdx.x * blockDim.x + threadIdx.x;
    if(i >= M || j >= M)
        return;
    float c = 0;
    for(size_t k = 0; k < N; n++)
        c += A[i*M+k] * B[k*M+j];
    C[i*M + j] = c;
}
```

⚠ Ma qui abbiamo aggiunto una **complessità**: se io mettessi B[j*M+k], potrebbe non funzionare se ho la memoria della GPU ordinata in row-major; se invece avessi column-major, sicuramente funziona. Quindi dobbiamo anche sapere come è ordinata/organizzata la memoria della GPU (le informazioni)

Different approaches of GPGPU programming:

- **GPU enabled libraries** → enable GPU acceleration without in-depth knowledge of GPU programming, but with high-quality implementations, tuned by experts
- **High level directives** (OpenACC, Thrust…) → added to serial source code to manage parallelization (es. loop); formatted as comments (don't interfere with serial execution), can be combined with explicit CUDA



OpenMP

```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

OpenACC

```
main() {
  double pi = 0.0; long i;

  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

printf("pi = %f\n", pi/N);
}
```

- **High level programming languages** (C/Cuda, OpenCL, GPU enabled libraries…) → **CUDA** (Computed Unified Device Architecture) allow to use GPUs for general-purpose programming; is a complete toolkit to compile, debug and run programs easily in a heterogeneous systems with GPUs. An alternative standard open-source programming model is **OpenCL**; is specialized also for other accelerations (not only GPU) [another framework is **Vulkan**]. Another (used by AMD) is **HIP**

CUDA:

```
__global__ void vector_add(const float* A,
                           const float* B,
                           float* C,
                           int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

HIP:

```
__global__ void vector_add(const float* A,
                           const float* B,
                           float* C,
                           int N) {
    int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

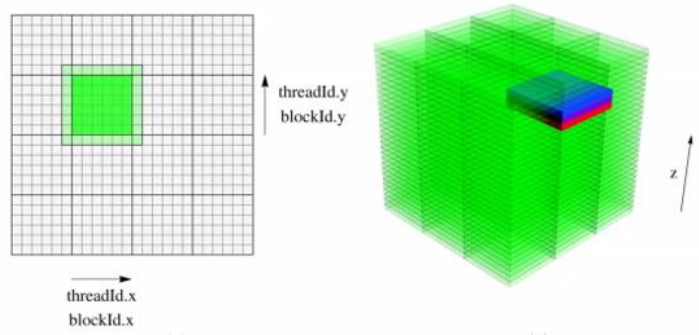
OpenCL:

```
__kernel void vector_add(__global const float* A,
                         __global const float* B,
                         __global float* C,
                         const int N)
{
    int i = get_global_id(0);
    if (i < N)
        C[i] = A[i] + B[i];
}
```

- **Low level programming languages** (PTX, assembly…)

4) Programming Fundamentals (CUDA)

How do I decide the right grid and block size (in CUDA)? What are warp divergence and thread divergence? How does memory hierarchy works?



Keywords in CUDA are:

- **Thread** → distributed by the CUDA runtime (`threadIdx`)
- **Block** → a user defined group of 1-1024 threads (`blockIdx`)
- **Grid** → a group of 1 or more blocks

For many parallelizable problems involving arrays, it's useful to think of *multidimensional arrays* (`dim3` is a struct to define your Grid and Block dimensions).

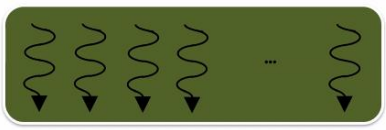
```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#if defined(__cplusplus)
    __host__ __device__ dim3(unsigned int vx = 1, unsigned int vy = 1, unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
    __host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
    __host__ __device__ operator uint3(void) { uint3 t; t.x = x; t.y = y; t.z = z; return t; }
#endif /* __cplusplus */
};
```

Example of **`vector_add()`** using **`blockIdx`** and **`threadIdx`**:

```
__global__ void add(int *a, int *b, int *c) {
   c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);
```

```
__global__ void add(int *a, int *b, int *c) {
   c[ThreadIdx.x] = a[ThreadIdx.x] + b[ThreadIdx.x];
}

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);
```

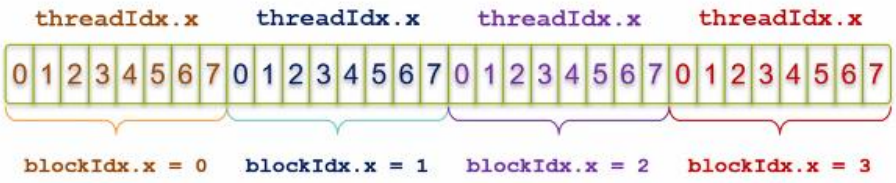
On the device, each block can execute in parallel:



With `M` (= `blockDim.x`) threads/block, a unique index for each thread is given by:

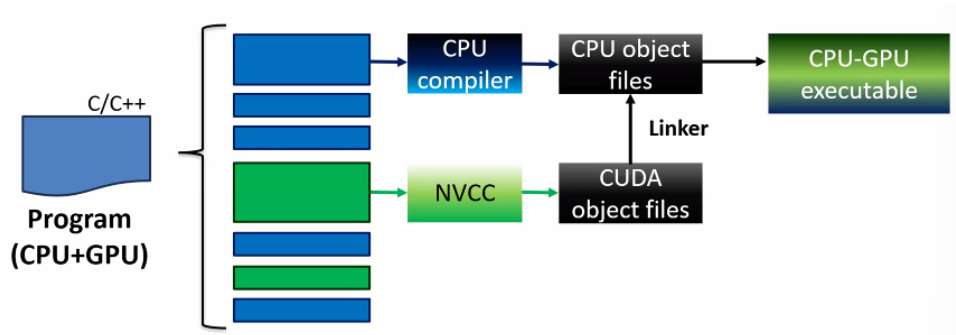
```
int idx = threadIdx.x + blockIdx.x * blockDim.x
```

so clearly > complexity:



→ CUDA

The programming stack for CUDA is:



- `cuda_runtime.h (-lcudart)`: provide the **runtime API** for CUDA programming, which allows developers to interact with and manage GPU resources in a more abstracted and user-friendly way compared to the lower-level driver API.
- **Device management:** cudaGetDevice(), cudaSetDevice(), and cudaGetDeviceProperties()
- **Memory management:** cudaMalloc(), cudaFree(), cudaMemcpy()
- **Kernel Launching:** triple angle bracket syntax (<<< >>>) for launching kernels
- **Error Handling:** Provides error codes and functions like cudaGetLastError

`cudaSetDevice()` defines a GPU to be used with its properties; used with **`cudaDeviceSynchronize()`** to synchronize the GPU in the code

`cudaMemcpy(void* dst, void* src, size_t count, cudaMemcpyKind)` copies memory between host (CPU) and device (GPU); **`cudaMemcpyKind`** can be `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`

The operator `<<<>>>` is unique to the CUDA runtime API and simplifies launching GPU kernels; the syntax is:

```
Kernel_name<<<numBlocks, threadPerBlock, streams(*), mem(*)>>>(in_out_args)
```

⚠ Ricorda che **`threadPerBlock = blockDim`**

Managing errors:

```
// Check for launch errors
cudaError_t err = cudaGetLastError(); // Returns the last error from
                                      a CUDA runtime call,
                                      including kernel launches.
if (err != cudaSuccess) {
    printf("Kernel launch failed: %s\n", cudaGetErrorString(err));
}
```

```
// Check for runtime errors
err = cudaDeviceSynchronize();        // returns any errors that
                                         occurred during execution.
if (err != cudaSuccess) {
    printf("Kernel execution failed: %s\n", cudaGetErrorString(err));
}
```

⚠ In realtà possiamo usare la macro **CUDA_CHECK** per controllare se avvengono errori:

```
#define CUDA_CHECK(call) \
    do { \
        cudaError_t err = call; \
        if (err != cudaSuccess) { \
            fprintf(stderr, "CUDA error in %s (%s:%d): %s\n", \
                    #call, __FILE__, __LINE__, cudaGetErrorString(err)); \
            exit(EXIT_FAILURE); \
        } \
    } while (0)

…
CUDA_CHECK(cudaMalloc(&d_a, size));
CUDA_CHECK(cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice));
```

Come visto nel laboratorio, si possono creare **cudaEvent_t** per avere timestamp e controllare i tempi:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

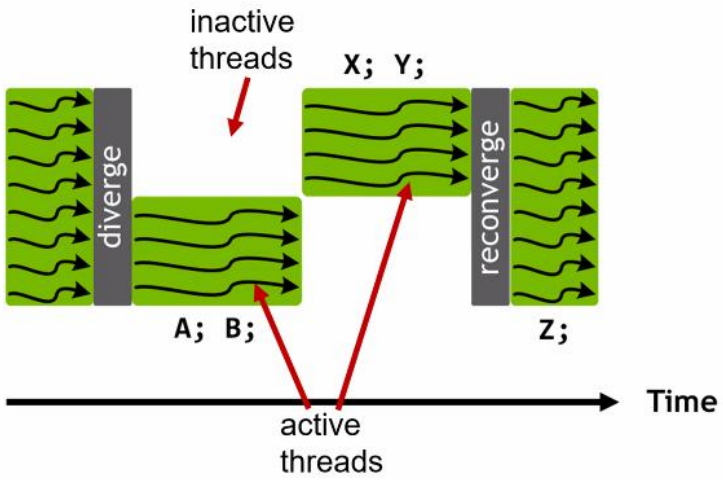
```
cudaEventRecord(start, 0);
// <<< kernel launches or GPU work >>>
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

Transparent scalability = ability of algorithm to operate on data of different sizes without reprogramming and user tuning (so launch more threads, then check for excess threads in the kernel) [una sorta di pruning sul numero di thread lanciati]:

```
__global__ void kernelFunc(float* A, float* B, size_t N)
{
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i >= N)
        return;
    … //do stuff
}
```

But here emerges the concept of **DIVERGENCE** (in base al numero di thread attuali, cambiare il flusso del programma) [le GPU moderne riducono il problema della divergence, aumentando le prestazioni]:

```
__global__ void foo(...)
{
…
    if (ThreadIdx.x < 4)
    {
        A;
        B;
    }
    else
    {
        X;
        Y;
    }
…
}
```



A way to avoid/reduce these *multi-path situations*, it's to operate on memory (on data) [> performance]; quindi in pratica si può usare DIVERGENCE, ma converrebbe trovare strade alternative!!!

When operating on more data (so on memories), thread synchronization is mandatory; using `__syncthreads()` allow to synchronize all the threads at block level (creando una ***barriera***) [quando lavoriamo in un sistema multi-GPU, si parla invece di *cooperative_groups* per la sincronizzazione]

A GPU has a max number of threads executable at same time; **OCCUPANCY** indicates the number of thread blocks that can be submitted to each SM during execution of the kernel (*resident blocks*). Occupancy can be controlled by varying compiler and launch parameters (blockDim, number of blocks, use of memories [shared]).

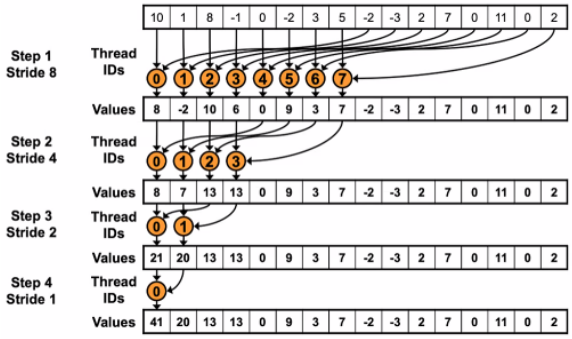
A function/kernel can be `__global__` or `__device__` (can't be called from HOST code [CPU]):

```
__global__ void calledFromCpuForGPU(...)
{
  //This function is called by CPU for execution on GPU
}

__device__ void calledFromGPUforGPU(...)
{
  // This function is called by GPU for execution on GPU
}
```

⚠ Best practices for CUDA:
- maximize occupancy
- Llmit thread divergence
- minimize global memory access
- use shared memory wisely
- align memory

EXAMPLE of a CUDA OPERATION (Parallel reduction)
A reduce operation takes in input an array and reduces it to a single value using a binary operation; in CUDA, **reduction** must be done in **parallel** (threads must coordinate to avoid race conditions, memory access must be efficient, synchronization is needed within blocks).

A strategy is **sequential addressing**:

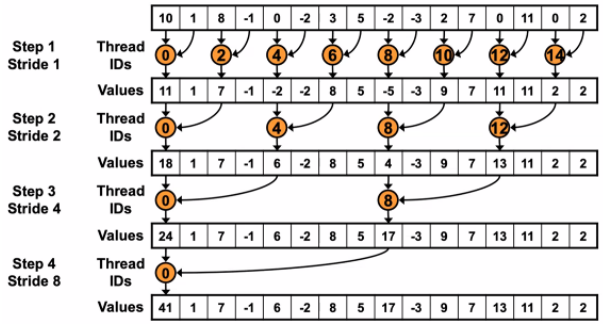


Ma come facciamo a dire in CUDA che i thread usati vanno dimezzati ad ogni step? Al primo step, basterebbe dire che per tutti gli elementi con threadIdx.x < N, la somma avverà tra l'elemento threadIdx.x e threadIdx.x + N/2; ma poi allo step dopo dovrò dimezzare ulteriorment. In codice:

```
__global__ void reduction_sum(const float *a, float *result) {
  int tid = threadIdx.x;                  // Thread ID within the block (0 - 1,023)
  int globalId = blockIdx.x * blockDim.x + tid;   // index
  int offset = 0;
  for (offset = blockDim.x / 2; offset > 0; offset = offset / 2) {
  // Rounds
     if (tid < offset) {
       a[tid] = a[tid] + a[tid + offset];
     }
  }
  if (tid == 0) {                  // Write the result from the first thread
      result = a[tid];      }     }
```

⚠️ Questa soluzione funziona però solo con array fino a 1024 elementi!

Another strategy is **interleaved addressing**:



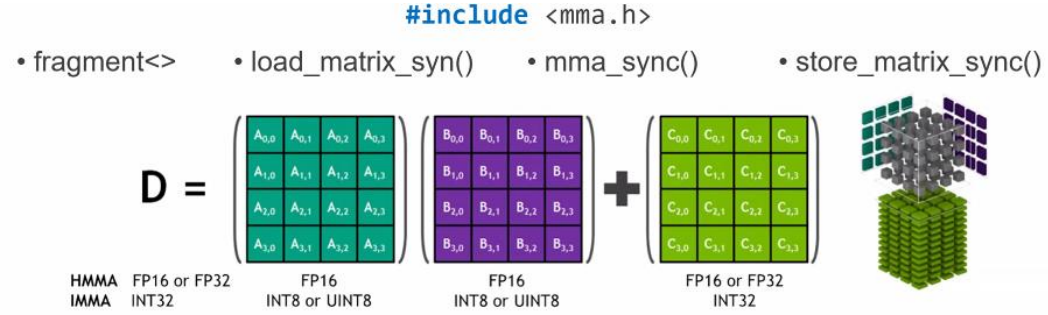
Il codice dobbiamo capirlo noi!!!

INTRINSICS are functions **provided by the compiler** that give access to **low-level instructions or hardware-specific operations without writing assembly** (`sinf(x)` = standard math library, `__sinf(x)` is intrinsic). The goal is speed (fewer GPU cycles), control (let the programmer decide between accuracy and speed) and determinism (always map the same instruction [predictable performance]), but at cost of slight precision loss (quindi se non ci serve assoluta precision, vanno benissimo).

Another choice can be inline assembly, that allow to embed **PTX** (Parallel Thread Execution; a virtual language) instructions directly in C++ kernel code [in fact PTX is NVIDIA's intermediate assembly language, which sits between the CUDA C++ source and the final machine code (**SASS**) that runs on the GPU; it gives the programmer *fine-grained control* over GPU instructions and GPU registers]:

```
asm volatile ("PTX_code" : output_operands : input_operands : clobbers);
```

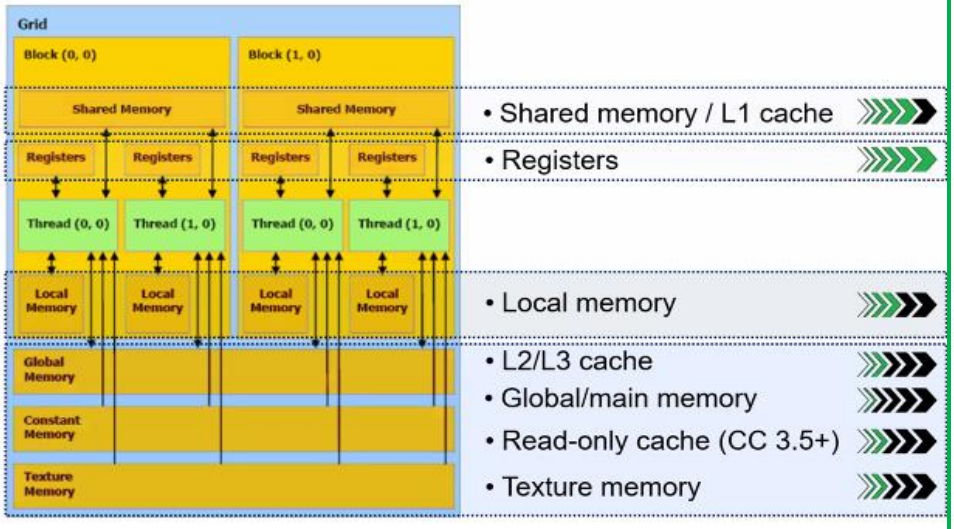
Special case of intrinsics is the tensor core case (programmable using intrinsics through WMMA API) which has warp granularity and not thread granularity:



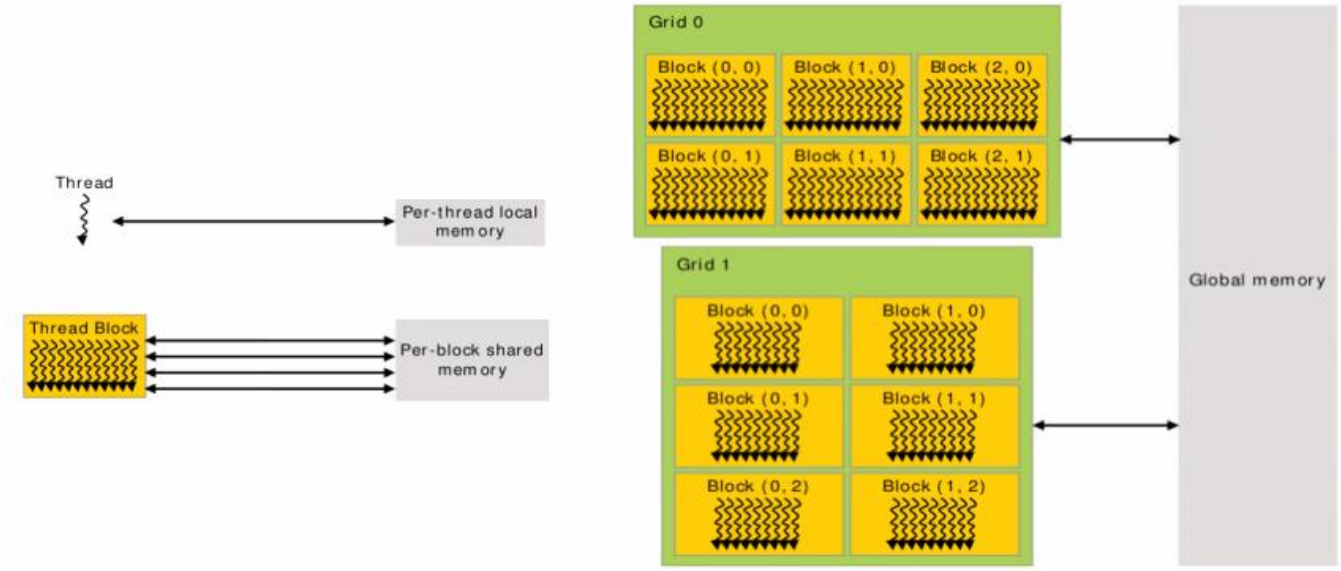
5) Memory Hierarchy

LATENCY = delay caused by the physical speed of hw; THROUGHPUT = max rate of production (data/s). CPU → low latency, low throughput; **GPU** → high latency, high throughput.

The target of the GPU memory hierarchy is provide data to the PEs and keep them busy, combining several memory layers to hide latency while providing flexibility.



Memory scope goes from single threads to blocks, to grids …):



Register = fastest form of memory; it's only accessible by the thread and has the lifetime of the thread. Each SM has 8-64k of 32-bit registers (*register pool*) [64/255 registers per thread]. Register allocation is defined by the compiler; hw scheduler and compiler attempts to avoid register *bank conflicts*

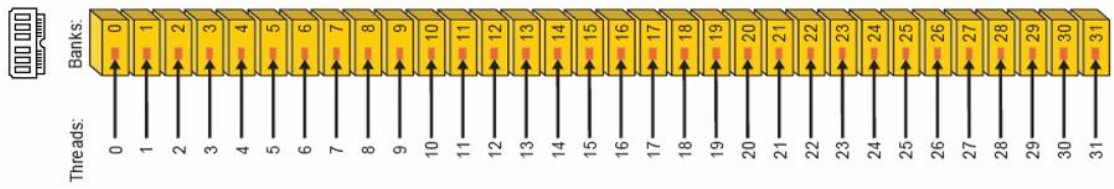
```
__global__ void foo(const float *A, const float *B, float *C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    …
    if (i < N) {
        // Using registers to store intermediate values
        float a = A[i];  // 'a' is stored in a register
        float b = B[i];  // 'b' is stored in a register
        float c = a + b; // 'c' is also stored in a register
        …
}
```

Shared Memory = can be as fast as a register when there are no *bank conflicts*. It's accessible by any thread of the block from which it was created and has the lifetime of the block. Shared memory size per SM is distributed across the blocks (requested shared memory per block influence the **occupancy** [*blocks per SM*])

```
__global__ void foo(float *input, float *output, int N) {
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float sharedData[256];      // Shared memory declaration

    if (i < N){
        sharedData[tid] = input[i];       // assignation per thread
    }
    __syncthreads();       // Ensure all threads have loaded their data
    …
    for (int s = blockDim.x / 2; s > 3; s >>= 2) {
        sharedData[tid] += sharedData[tid + s];
        __syncthreads();
    }
    …
output[blockIdx.x] = sharedData[0];
}
```

⚠ Some GPUs can configure the caches (L1 Cache, Texture Cache), some can configure memory banks (each bank can be accessed independently; consecutive 32-bit words are in consecutive banks; bank *conflicts* are serialized [except for reading the same address (broadcast)]).

Linear addressing (ideal) → each thread in a warp access different memory bank; no collisions; in reality, the **programmer must guarantee mostly linear addressing on the shared memory**.



Local Memory = resides in main/global memory and can be 150x slower than register or shared memory; is only accessible by the thread and has the lifetime of the thread. Useful to store arrays that can't be placed in registers (`float var[local_memory_size]`). It's automatically allocated by compiler and release register pressure [optimizing CUDA usually involves also minimizing reliance on local memory, because is slower]

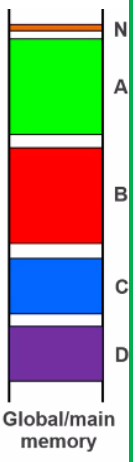
```
__global__ void localMemoryExample(float *output, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < N) {
        // This array is private to each thread, but stored in local memory
        float localArray[4];
        // This array is also private to each thread, but stored in local memory
        char large_Array[400];
        …
}
```

Costant Memory = special 64KB cache for read-only data (all threads in a grid); helpful to broadcast data among all threads in a kernel (but obviously slower); sometimes it's also called **Texture Memory** (fully-specialized for 2D special locality)

```
__constant__ int d_lookupTable[256];     // constant memory array in GPU
…
__global__ void Foo(int *output) {       // Kernel that uses constant memory
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {                       // Using values from lookup table
      output[idx] = d_lookupTable[idx % 256];
    }
}
…

int main() {
int h_lookupTable[256];
int h_output[N];
for (int i = 0; i < 256; ++i){           // Initialize lookup table
  h_lookupTable[i] = i * i;              // square values
}
//   Copy lookup table to constant memory
cudaMemcpyToSymbol(d_lookupTable, h_lookupTable, 256 * sizeof(int));
…
}
```

Global/Main Memory = separate hw from GPU core (containing SM's L2/L3 caches); really slower than register and shared memory; accessible from both the host or device and has the lifetime of the application (is **persistent between kernel launches**)

```
__global__ void foo(float *A, float *B, float *C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float var = 2.34;
    …
    if (i < N) {
    …
}

__global__ void bar(float *C, float *D, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float w[4] = {1.23, 2.99, 3.3, 4.09};
    …
    if (i < N) {
    …
}

…

foo<<blocks_per_grid, threads_per_block>>>(d_A, d_B, d_C, N);
bar<<<blocks_per_grid, threads_per_block>>>(d_C, d_D, N);
```



So **HOW TO CONFIGURE GLOBAL MEMORY**? A programmer can configure the global/main memory access in 1 of 4 main strategies:

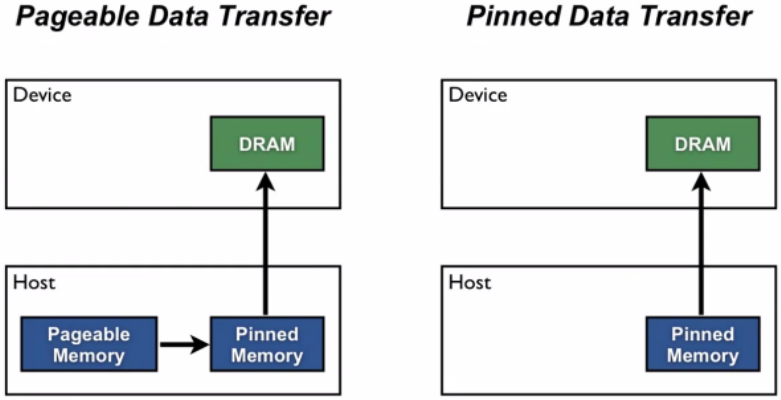
- **Pageable** (*default* config) → host-device transfers: regular data is allocated on the host; not directly accessible by GPU; requires explicit copying with `cudaMemcpy(dst, src, size, direction)` to transfer data host-device. Mostly used in parallel applications where performance is not critical:

```
…
float *h_A = (float *)malloc(size);                    // Allocate paged memory on host
…
float *d_A, *d_B, *d_C;
cudaMalloc((void **)&d_A, size);                       // Allocate memory on device
…
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);    // Copy data from host to device
…
foo<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
…
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);    // Copy result back to host
…
free(h_A);                                             // Free host memory
…
cudaFree(d_A);                                         // Free device memory
```

Overlapping: **`cudaMemcpyAsync(dst, src, size, direction, Stream)`** is asynchronous for the host, so that we can start an operation on the host before the copy on GPU is completed to wait less

```
float *h_data;                                                  // Allocate pinned host memory
cudaMallocHost((void**)&h_data, size);                          // (Required for async copy)
…
float *d_data;
cudaMalloc((void**)&d_data, size);                              // Allocate device memory
…
cudaMemcpyAsync(d_data, h_data, size, cudaMemcpyHostToDevice);  // Asynchronous copy to device
…
bar<<<blocksPerGrid, threadsPerBlock>>>(d_data, N);             // Launch kernel
host_func();                                                    // useful host functions…
…
cudaMemcpyAsync(h_data, d_data, size, cudaMemcpyDeviceToHost);  // Asynchronous copy back to host
cudaDeviceSynchronize();                                        // Synchronize to ensure all operations
                                                                // are complete
cudaFreeHost(h_data);                                           // Clean up
cudaFree(d_data);
```

- **Pinned** (*page-locked memory*) → memory stays in **physical RAM**, which allows for faster and efficient data transfers between host and device (using DMA [direct memory access], supporting `cudaMemcpyAsync` and a more efficient bandwidth usage)

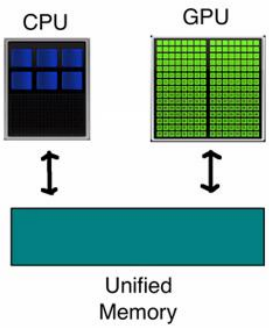



```
h_Pageable = (float*)malloc(bytes)    cudaMallocHost((void**)&h_Pinned, bytes)
```

- **Mapped** (*zero-copy memory*) → leverage host memory when there is insufficient device memory (large data sets); useful for app where data changes frequently on the host. It avoid explicit data transfers host-device: data transfers **occur during GPU execution**, which increase the processing time; requires **synchronization**; slower than pageable

```
…
cudaSetDeviceFlags(cudaDeviceMapHost);                  // Allocate pinned host
                                                        // memory that is mapped to device
cudaHostAlloc((void**)&hostData, size * sizeof(int), cudaHostAllocMapped);
…
cudaHostGetDevicePointer((void**)&deviceData, (void*)hostData, 0); // Get device pointer to
                                                                   // the mapped host memory

addKernel<<<blocksPerGrid, threadsPerBlock>>>(deviceData, 10, size);
cudaDeviceSynchronize();
for (int i = 0; i < 10; ++i) {                                  // Print some results
    std::cout << "hostData[" << i << "] = " << hostData[i] << std::endl;
}
cudaFreeHost(hostData);                                         // Free memory
```

- **Unified** → **single shared memory for host (CPU) and device (GPU)** [so not `cudaMemcpy`]; allows data access without needing to explicitly copy it back and forth; simplifies CUDA for multi-GPU systems. So it has automatic data migration, but limited performance (useful for prototyping, complex data structures and large datasets)




```
__global__ void bar (int *a, int *b, int *c, int n) {
    …
}
int main() {
    …
    int *a, *b, *c;                                  // Allocate unified memory
    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);
    …
    bar<<<blocksPerGrid, threadsPerBlock>>>(a, b, c, n);
    cudaDeviceSynchronize();                         // Wait for GPU to finish
    for (int i = 0; i < n; ++i) {                    // Verify results
        std::cout << "Error at index " << i << ": " << c[i] << " != " << a[i] + b[i] << std::endl;
    }
    …
    cudaFree(a);                                     // Free memory
    cudaFree(b);
    cudaFree(c);
    return 0;
}
```

Global memory vs Shared memory:

- Global memory:
  - **Issue** = data race conditions
  - **Coalescing** = threads in same warp access memory simultaneously, optimizing memory access (reduce # of accesses and speed up data transfer [cache hits])
  - **Alignment** = optimal data organization in memory
- Shared memory:
  - **linear addressing**:
    - **Issue** = bank conflicts
    - Each thread in same warp access different memory bank (id fully independent)
    - **No collisions**
  - **linear addressing with stride**:
    - **2-way conflict**: each thread access 2^{ith} item (or 3^{ith} etc…)
    - **No collisions**

⚠ In older shared memory (*CC 1.x*) broadcast is served independently, in *CC 2.x*, simultaneously

So what are **solutions** for bank conflicts in SHARED MEMORY?

- **PADDING** = add 1+ empty bytes between the different data types so that accesses don't map to the same shared memory bank:

```
__global__ void kernel_With_Padding(float *input, float *output, int N)
{
    // Shared memory with padding: Padding of 1 extra float for bank conflicts
    __shared__ float sharedMem[N][N + 1];
    // Calculate row and column indices
    int row = threadIdx.y;
    int col = threadIdx.x;
    // Load input into shared memory with padding
    sharedMem[row][col] = input[row * N + col];
    __syncthreads();
    // Example operation: transpose matrix
    output[row * N + col] = sharedMem[col][row];
}
```

- **SWIZZLING** = same as padding, but for rearranges the mapping of the shared memory index to avoid/reduce shared memory bank conflicts (used in matrix operations)

6) Programming Fundamentals in CUDA

ATOMICS = special operations that are guaranteed to be indivisible (executed **atomically**); in multi-thread systems, **atomics compute each thread in a sequential fashion**, preventing race conditions where multiple threads try to access and modify the same memory location simultaneously (but performance cost due to the serialization of operations). So atomics are essential for using shared variables (shared memory) and implementing complex lock-free data structures. Let's see an example of not atomic vs atomic to increment a shared counter through many threads:

```
__global__ void count_kernel(int *counter) {
    (*counter)++;          // Not atomic!
}
```

```
__global__ void count_kernel_atomic(int *counter) {
    atomicAdd(counter, 1);
}
```

Supported formats and operations of atomics →

Integers
- int
- unsigned int
- unsigned long
- long int

Floating point
- float
- double

Operation	Function	Description
Addition	atomicAdd(address, val)	Adds val to *address
Subtraction	(use atomicAdd(address, -val))	
Min/Max	atomicMin, atomicMax	Updates with min/max
Exchange	atomicExch(address, val)	Writes val and returns the old value
Compare and Swap	atomicCAS(address, compare, val)	If *address == compare, replace with val
Bitwise	atomicAnd, atomicOr, atomicXor	Bitwise operations

⚠ Ovviamente non basta la **`__syncthreads()`** se sto accedendo e provando a modificare la stessa variabile/ zona di memoria da più thread insieme (*race condition*)

⚠ Riprendendo la parallel reduction (che avevamo usato per calcolare la somma cumulativa degli elementi di un array), se usiamo il block per fare lo store del result temporaneo (mettendo il risultato nella cella del block con a[tid]), avremmo come **limite** che può avvenire parallelamente solo su 1024 threads (ovvero il n° max di threads per block). So, the solution is to use SHARED MEMORY and atomics:

```
__global__ void reduction_sum(const float *a, float *result) {

    int tid = threadIdx.x;                          // Thread ID within the block
    int globalId = blockIdx.x * blockDim.x + tid; // Load data into shared memory
    extern __shared__ float sharedData[];          // Dynamically allocated shared memory
    sharedData[tid] = a[globalId];                 // copy from global to shared memory
    __syncthreads();
    for (int offset = blockDim.x / 2; offset > 0; offset = offset /2) {
       if (tid < offset) {
         sharedData[tid] = sharedData[tid] + sharedData[tid + offset];
       }
       __syncthreads();                          // Synchronize to ensure data is consistent
    }
    if (tid == 0) {                 // Write the result from the first thread of each block
          atomicAdd(result, sharedData[0]);
    }
    }
```

In pratica nel codice si vede che in **`sharedData[0]`** finisce il risultato locale del singolo block; con la **`atomicAdd`** al fondo si sommano questi risultati parziali e si ottiene il risultato finale (questa `atomicAdd` ovviamente è seriale, ma l'accumulo locale al blocco avviene parallelamente per ogni blocco).

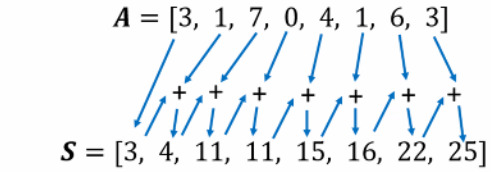
Con **`extern`** si permette di allocare dinamicamente la dimensione di `sharedData` (che può quindi essere chiamato dall'host quando chiama il kernel con il parametro che serve in quel momento passato insieme ai parametri threadPerBlock etc… all'interno della chiamata del kernel in **`<<<>>>`**).

⚠️ Quella che abbiamo appena visto è anche nota come "prefix sum" (una primitive algoritmica parallela fondamentale, usata in GPU/FPGA design, parallel computing e deep learning accelerators).
Una **SCAN operation** produce un vettore di output dove ogni elemento è un'operazione sugli elementi dell'input fino ad un certo punto del vettore in input (**se questa operazione è la somma abbiamo la prefix sum**). Vediamo differenza tra **inclusive** and **exclusive** prefix sum:

- Inclusive prefix sum:

$$S_i = \sum_{j=0}^{i} a_j$$

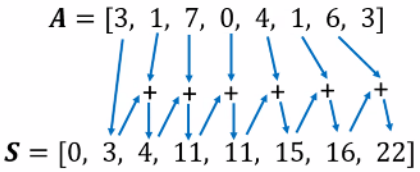
Example:



- Exclusive prefix sum:

$$S_i = \sum_{j=0}^{i-1} a_j, \qquad S_0 = 0$$

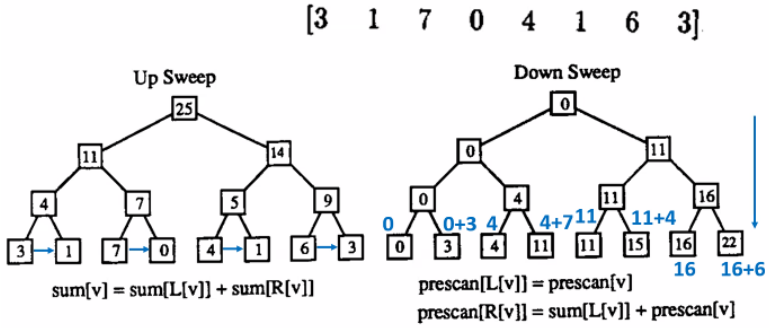
Example:



An exclusive scan can be generated from an inclusive scan by shifting the resulting array right by one element and inserting the identity

To implement a SCAN operation, we have 2 phases to do ($O(\log n)$):

- UPSWEEP (reduction) → build a binary tree of partial results; each node stores the result of its 2 children; the final root holds the final/total result
- DOWNSWEEP → propagate partial sums downward to compute the scan operation for each node; each node receives the accumulated result from the left subtree



In the ideal implementation of scan, we assumes that system has as many processors as data elements (so no the case of GPUs) and we have to overwrite the accumulation array. So the **real implementation**:

```
__global__ void scan(float *g_odata, float *g_idata, int n) {
  extern __shared__ float temp[];     // allocated on invocation
  int thid = threadIdx.x;
  int pout = 0, pin = 1;              // double buffer indices
  // This is exclusive scan, so shift right by one and set first element to 0
  temp[pout * n + thid] = (thid > 0) ? g_idata[thid - 1] : 0;
  __syncthreads();
  for (int offset = 1; offset < n; offset = offset*2) {   // d
    pout = 1 - pout;                  // swap double buffer indices
    pin = 1 - pout;
    if (thid > = offset)
      temp[pout * n + thid] = temp[pout * n + thid] + temp[pin * n + thid - offset];
    else
      temp[pout * n + thid] = temp[pin * n + thid];
    __syncthreads();
  }  // for()
// write output
g_odata[thid] = temp[pout * n + thid];
}
```

Final scan storage from shared mem to global mem

⚠️ Dal nostro punto di vista, noi useremo queste scan operations già implementate in progetti (sono già fatte, noi dobbiamo capire come usarle). Infatti noi abbiamo dei TEMPLATES ↓

TEMPLATES in CUDA/C++ are a powerful feature that allows us to **write generic and reusable code** (tipo interfacce); they enable functions and classes to operate with generic types:

```
template <typename T>
__global__ void addKernel(const T* a, const T* b, T* c, int n) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n)
        c[i] = a[i] + b[i];
}
...
addKernel<float><<<blocks, threads>>>(d_a_f, d_b_f, d_c_f, N);
addKernel<double><<<blocks, threads>>>(d_a_d, d_b_d, d_c_d, N);
...
```

CUDA templates allow the compiler to generate specialized code at compile time (also enabling full device-side optimization). Also parameterization by constants, using **non-type template parameters** (like tile sizes or block dimensions) **to tune performance** at compile time:

```
template <int T_size>
__global__ void reduceKernel(float* data) {
    __shared__ float Data_M[T_size][T_size];
    ...
}
```

```
…
reduceKernel<128><<<...>>>(data);
reduceKernel<256><<<...>>>(data);
…
```

⚠️ Infatti avviene *training* sulle librerie soprattutto al fine di trovare le configurazioni migliori per le prestazioni quando si lancia il compile

CUDA allows templated `__host__ __device__` functions (same template can be used by CPU + GPU):

```
template <typename T>
__host__ __device__ T square(T x) {
    return x * x;
}
```

→ **PROJECT ASSIGNMENT**: apply data parallelism and CUDA concepts; optimize performance in deployment of algorithms/app on GPU; develop optimized libraries for GPU; develop system using GPU. Focus anche su comunicazione e esposizione del lavoro fatto. Si possono fare anche ricerche su lavori già esistenti o in generale ricerche su un topic legato alle GPU. Regole:

Proposal

- **Date:** Submitted at least 6 weeks before the presentation day
- Winter session exams: **01/02/2026 – 22/02/2026**
- Summer session exams: **08/06/2026 – 18/07/2026**
- 1 page with an abstract of the project:
  must include:
  - Title of the project
  - Authors
  - Main problem
  - Targets or objectives of the assignment
  - References

Deadline	Exam dates (4 sessions)
December 24th / 2025	e.g., February 4th
January 6st	e.g., February 17th
April – May	June – July
July	September 1st – 20th

- If you miss the winter deadlines, you cannot apply for project presentation in those sessions
- The same happens in case of a change of the project

Report

- **Date:** submitted at **least one week before the presentation day** (e.g., winter, summer exam session)
- 10-12 pages, IEEE 2-column, times new roman, 10pf
  https://www.ieee.org/conferences/publishing/templates
  Must include: (don't be verbose or superficial)
  - Introduction, brief background, methodology / proposed method
  - Experimental results and analysis of results (meaningful figures and pictures)
  - Discussion of the results
  - References
- Technically sound, clear, organized, and well-written

Presentation

- **Date:** Slides and project codes submitted at least one week before the presentation day
- **Time:** 15 mins + 5 mins Q/A
- don't be verbose or superficial
- **Functionality and performance:**
  - Produces correct results
  - Achieves good speedup relative to base code and/or competitors
  - Optimizations applied
- **Code Quality: (GitHub link is strongly advised)**
  - Clear coding
  - Well documented

Intermediate check:

Starting from December 17, we will organize roundtable activities
- Focused on specific group activities
- Everyone can join the discussion, share ideas, propose solutions…

We will identify fixed slots with the groups
- We will interact with you and allocate them according to everyone's availability

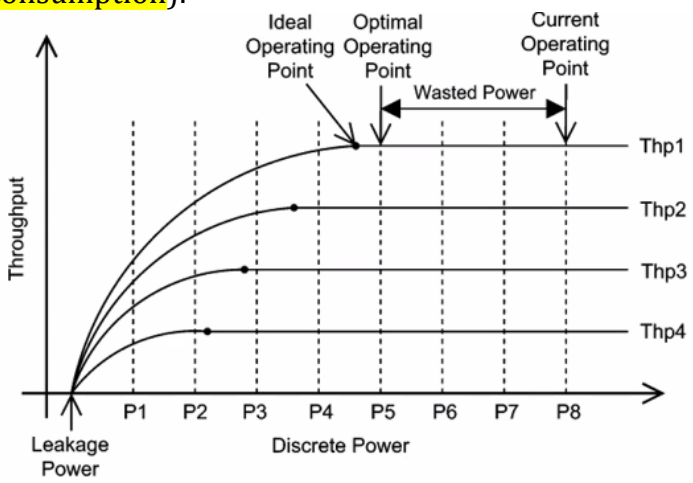
If you want to apply for the winter exam sessions, you must attend at least one of the intermediate checks

Example of application optimization:

Step	Brief outline	Concrete example
1	Choose an application	Dense Matrix-Matrix Multiply
2	Determine what part of the application is taking the majority ofthe time	Algorithm Evaluation through Analysis/Simulation
3	Determine one or more data-parallel approaches to solving the problem	Assign one output to each thread in a gather-style approach
4	Create multiple implementations of the approach	One naïve version, one version with shared memory tiling, one version with register tiling
5	Measure the performance and execution characteristics of the implementations for various parameters	Record memory transfer time, kernel time, utilization, FLOPS, etc
6	Relate results to course concepts	Performance may be impacted by utilization, shared-memory accesses, memory coalescing, and control divergence.

7) Profiling

The goal of **PROFILING** allows to monitor how the resources of the system are used (so profiling != debugging) enabling us to understand and optimize the performance of our application (monitoring both the host [CPU] and the device [GPU]), also **identifying bottlenecks** (l'obiettivo è come sempre > throughput con < power consumption):



What **profiling infrastructures** are in GPU? An important one that we can use is the STATUS Register. So the main target of profiling is the observation of main features of an app on the running platform, using *performance* metrics (bandwidth, throughput, frequency, IPC, execution time) + *system* metrics (resources, instructions, parallel behavior) + *power* metrics.

⚠ Quindi in pratica ci interessa capire:
- **Memory utilization** (bandwidth, cache hits/misses)
- **Compute utilization** (SM occupancy, thread efficiency)
- **Kernel execution and power consumption**

Different **methods** can be used for profiling:
- in-system → profiler host and device are connected, so direct profiling
- remote → profiler host and device communicate externally through port
- command-line → profiler host interacts with the device
- custom development

Nvidia provides several tool to do profiling and optimization, but we will use **Nsight Systems** (for app profiling across GPU and CPU; runs on the Linux host computer) [**ncu** command]
⚠ Nel corso, se usiamo Jetson Nano [HPC remote], si usa **nvprof** (che è un command-line profiler tool)

Esempio di schermata di nsight (ncu) [solo la 1^ sezione, ce ne sono molte altre]:

```
Section: GPU Speed Of Light Throughput
---------------------------------------------------------------------- --------------- ------------------------------
DRAM Frequency                                                         cycle/nsecond                           6,99
SM Frequency                                                           cycle/nsecond                           1,41
Elapsed Cycles                                                                 cycle                      3.624.887
Memory [%]                                                                         %                          97,58
DRAM Throughput                                                                    %                          16,42
Duration                                                                      msecond                           2,57
L1/TEX Cache Throughput                                                            %                          97,95
L2 Cache Throughput                                                                %                          19,96
SM Active Cycles                                                               cycle                      3.611.201,97
Compute (SM) [%]                                                                   %                          97,58
---------------------------------------------------------------------- --------------- ------------------------------
INF   The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To
      further improve performance, work will likely need to be shifted from the most utilized to another unit.
      Start by analyzing workloads in the Compute Workload Analysis section.

INF   The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 6% of
      this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide
      (https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline) for mode details on roofline
      analysis.
```

⚠ Se voglio una versione più capibile con interfaccia grafica è con **ncu-ui** (stesso comando, ma usa UI)

8) Streams, algorithms and applications

If kernel *A* and kernel *B* are independent, why force the GPU to run $A \rightarrow B$ and not ***A*** || ***B***? In this part we talk about **concurrency** (when **2 or more CUDA operations proceed at the same time**) nd **dynamic parallelism** in GPUs.

To **obtain CONCURRENCY**:
- operations must be assigned to streams different from the default stream
- host/device data trasfers should be asynchronous and host memory must be page-locked

⚠ *N*-way concurrency indica che in 1 clock si possono eseguire in parallelo in pipeline *N* operations

Every CUDA action is submitted to an execution queue on the device (GPU); CUDA runtime functions can be so **blocking** (*synchronous*) and **non-blocking** (*asynchronous*) [CUDA API provides asynchronous versions of synchronous ones]
⚠ If we use asynchronous we have to manage the synchronization, but we can also have concurrency (overlap computation on host and device; execute more than 1 kernel on same device; data transfers between host and device while executing a kernel, or from H2D while transferring data D2H)

- **blocking** (synchronous):
  return control to ***host*** after execution is completed on ***device***
  - all memory transfer > 64KB
  - all memory allocation on *device*
  - allocation of page locked memory on *host*
- **Non-blocking** (asynchronous):
  return control to ***host*** **immediatelly**, while its execution proceeds on ***device***
  - all kernel launch are asynchronous
  - all memory transfers < 64KB
  - memory initialization on *device* (**cudaMemset**)
  - memory copies from ***device*** to ***device***
  - explicit asynchronous memory transfers

GPU operations are implemented in CUDA using execution queues (called **STREAMS**):
- any operation pushed in a stream will be executed only after all other operations in same stream are completed (FIFO queue)
- operations assigned to different streams can be executed in any order with respect to each other
- CUDA runtime provides a default stream (***stream 0***) if not explicitly declared

⚠ CUDA Streams advantages: overlapping of operations, hide memory latency, utilize better GPU resources and control over scheduling; disadvantages: > complexity, debugging more difficult, limited benefits for workloads with high data interdependency or that already fully utilize GPU resources

Synchronization of CUDA Streams:
- Explicit
  - `cudaDeviceSynchronize()`
    - Blocks host code until all operations on the device are completed
  - `cudaStreamSynchronize(stream)`
    - Blocks host code until all operations on a stream are completed
  - `cudaStreamWaitEvent(stream, event)`
    - Blocks all operations assigned to a stream until event is reached
- Implicit (all operations assigned to default stream; all page-locked memory allocations; all memory allocations on device; all settings operation on device)

Managing CUDA Streams (`CudaStream_t` is the type of CUDA Streams):

- constructor → **`cudaStreamCreate()`** or **`cudaStreamCreateWithFlags(, flag)`** [this can be default or non-blocking (indicates that the created stream doesn't synchronize with the default stream)]; another one is **`cudaStreamCreateWithPriority()`** (to specify the priority among the streams)
- synchronize → **`cudaStreamSynchronize()`**
- destructor → **`cudaStreamDestroy()`**

Example

```
cudaSetDevice(0);

cudaStreamCreate(stream1);
cudaStreamCreate(stream2);

// concurrent execution of the two equievalent kernels
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp_1, out_1);
Kernel_1<<<blocks, threads, SharedMem, stream2>>>(inp_2, out_2);

// concurrent execution of different kernels
Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp, out_1);
Kernel_2<<<blocks, threads, SharedMem, stream2>>>(inp, out_2);

cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
```

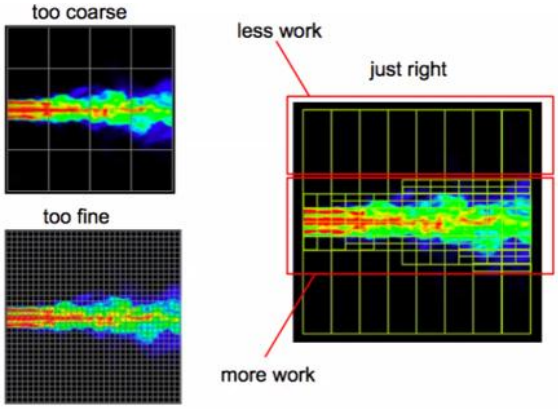
`CudaMemcpyAsync()` performs explicitly asynchronous data transfers between host and device memory; data transfers should be queued into a stream different from the default one to be asynchronous

Example

```
cudaStreamCreate(stream_a);
cudaStreamCreate(stream_b);
cudaMallocHost(h_buffer_a, buffer_a_size);
cudaMallocHost(h_buffer_b, buffer_b_size);
cudaMalloc(d_buffer_a, buffer_a_size);
cudaMalloc(d_buffer_b, buffer_b_size);

// synchronous and concurrent transfer H2D and D2H
cudaMemcpyAsync(d_buffer_a, h_buffer_a, buffer_a_size, cudaMemcpyHostToDevice, stream_a);
cudaMemcpyAsync(h_buffer_b, d_buffer_b, buffer_b_size, cudaMemcpyDeviceToHost, stream_b);

cudaStreamDestroy(stream_a);
cudaStreamDestroy(stream_b);

cudaFreeHost(h_buffer_a);
cudaFreeHost(h_buffer_b);
```

⚠ Streams also help in big systems with more GPUs (where you can access the single GPU through gpu_id); also, using CUDA API, you can access *properties* of the device to manage it!

Adaptive Parallel Computation (***nesting kernels***) = in the past programs mostly performed a sequence of kernel launches and for best performance each kernel had to expose enough parallelism to use efficiently the GPU. An example of nested parallelism are *adaptive grids* (img dx) (es. in videogiochi dove elementi lontani sono rappresentati in modo semplificato per ridurre la computazione e vengono renderizzati meglio quando ci si avvicina)

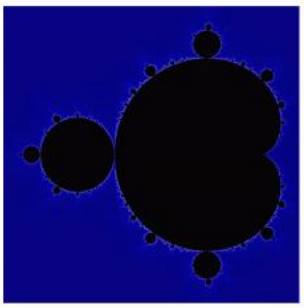


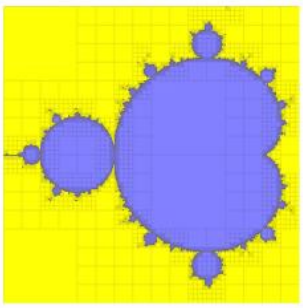
Dynamic Parallelism = **launch kernels from threads running on the device**

Example (***Mandelbrot set***) → a point c belongs to the Mandelbrot set if the sequence doesn't blow up to infinity:

- if $|z_n|$ stays bounded (never gets very large), the point is inside the set
- if $|z_n|$ grows without limit, the point is outside the set

Instead of proceeding point by point, we can use the *adaptive grids* (dynamic parallelism) to reduce the computation for unuseful points (*mariani-silver algorithm*):





```
mariani_silver(rectangle)
  if (border(rectangle) has common dwell)
        fill rectangle with common dwell
    else if (rectangle size < threshold)
      per-pixel evaluation of the rectangle
    else
      for each sub_rectangle in subdivide(rectangle)
          mariani_silver(sub_rectangle)
```

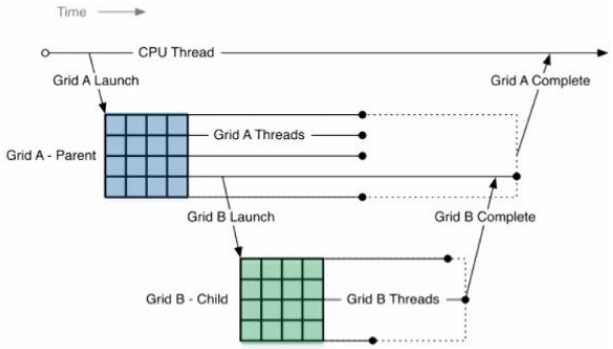
```
__global__ void mandelbrot_block_k(int *dwells, int w, int h, complex cmin, complex cmax,
                                   int x0, int y0, int d, int depth) {
  x0 += d * blockIdx.x, y0 += d * blockIdx.y;
  int common_dwell = border_dwell(w, h, cmin, cmax, x0, y0, d);
  if (threadIdx.x == 0 && threadIdx.y == 0) {
    if (common_dwell != DIFF_DWELL) {                          // uniform dwell, just fill
      dim3 bs(BSX, BSY), grid(divup(d, BSX), divup(d, BSY));
      dwell_fill<<<grid, bs>>>(dwells, w, x0, y0, d, comm_dwell);
    }
    else if (depth + 1 < MAX_DEPTH && d / SUBDIV > MIN_SIZE) {  // subdivide recursively
      dim3 bs(blockDim.x, blockDim.y), grid(SUBDIV, SUBDIV);
      mandelbrot_block_k<<<grid, bs>>>(dwells, w, h, cmin, cmax, x0, y0, d / SUBDIV, depth+1);
    }
    else {                                                     // leaf, per-pixel kernel
      dim3 bs(BSX, BSY), grid(divup(d, BSX), divup(d, BSY));
      mandelbrot_pixel_k<<<grid, bs>>>(dwells, w, h, cmin, cmax, x0, y0, d);
    }
    cucheck_dev(cudaGetLastError());
  }
} // mandelbrot_block_k
```

Only one thread starts the kernel call

https://github.com/canonizer/mandelbrot-dyn

⚠ Alla fine il dynamic parallelism è come una *sorta di ricorsione tra thread* (thread che chiamano da sè altri thread etc...)

Considerazioni finali sul dynamic parallelism:

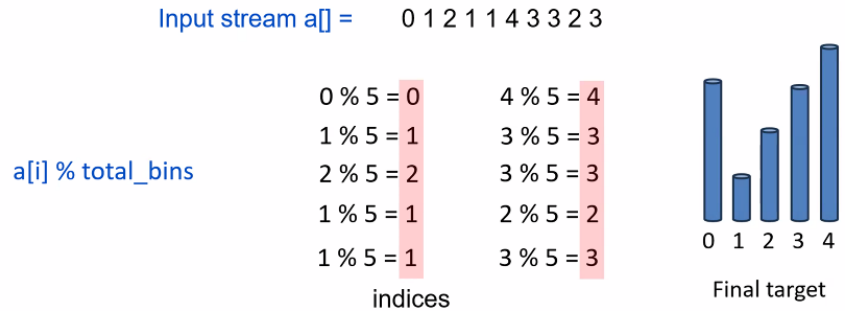


- **Grid and Synchronization**
  - parent grid (kernel) launches child grids (kernels)
  - child grids inherits from the parent grid certain attributes and limits
  - control flow for launching is mandatory
- **Memory consistency** (full consistency)
  - parent grid often relies on a child grid reading from and writing to global memory
- **Passing pointers to childs** (*arguments*)
  - can pass global memory (__device__ variables), allocated memory and constant memory
  - can't pass shared memory (__shared__ variables), local memory (and stack variables)
- **Device streams** → a child grid launched within a thread block are executed sequentially (FIFO); all streams created on the device are non-blocking
- **Recursion depth and device limits**

- **nesting depth** = deepest nesting level of recursive grid launches, with kernels launched from the host having depth 0
- **synchronization depth** = deepest nesting level at which `cudaDeviceSynchronize()` is called

Most cases **synchronization depth = nesting depth – 1**

Example → **Parallel Histograms**: classify and determine a histogram from input data using CUDA (input = array). Usati per le immagini o computer vision. Main strategy: module operators to define bins count:




```
__global__ void histogram(const int *input, int *histogram, int n, int numBins) {
  // Each thread processes one element
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid < n) {
    int bin = input[tid] % numBins;              // Compute the bin index
    atomicAdd(&histogram[bin], 1);               // Safely update of the histogram
  }
}
```

⚠ Ma se nell'effettivo l'immagine ha talmente tanti pixel (molto grossa) che non bastano i thread per la chiamata del kernel? Potremmo costruire local histograms usando atomics e poi mergiare questi local histograms for the final result; il local histogram ha questi parametri:

```
__global__ void local_his(pixel *in, int width, int height, unsigned int *out) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;      // pixel coordinates
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int nx = blockDim.x * gridDim.x;                    // grid dimensions
  int ny = blockDim.y * gridDim.y;
  int gl = blockIdx.x + blockIdx.y * gridDim.x;       // linear block index within 2D grid
  int t = threadIdx.x + threadIdx.y * blockDim.x;     // linear thread index within 2D block
  int nt = blockDim.x * blockDim.y;                   // total threads in 2D block
```


```
// initialize temporary accumulation array in global memory
  unsigned int *gmem = out + gl * NUM_PARTS;
  for (int i = t; i < 3 * NUM_BINS; i += nt)
        gmem[i] = 0;
// process pixels
  // updates our block's partial histogram in global memory
  for (int col = x; col < width; col += nx)
    for (int row = y; row < height; row += ny) {
      unsigned int r =  (unsigned int)(256 * in[row * width + col].x);
      unsigned int g = (unsigned int)(256 * in[row * width + col].y);
      unsigned int b = (unsigned int)(256 * in[row * width + col].z);
      atomicAdd(&gmem[NUM_BINS * 0 + r], 1);
      atomicAdd(&gmem[NUM_BINS * 1 + g], 1);
      atomicAdd(&gmem[NUM_BINS * 2 + b], 1);
    }
}
```

⚠ Potremmo migliorare ulteriormente le performance come al solito passando da global a shared mem!

9) Sorting in CUDA

SORTING is one of the most important algorithmic building blocks (primitives) and is a critical operation for large amounts of data. Sorting categories:

- data-driven → fastest algorithms, but bad performance if the sequence to sort is already sorted
- data-indipendent → algorithms don't change their processing and can be used in GPU:
  - **Bitonic Sort** = sorts data within a block using shared memory; for small/medium arrays; 1 block per array segment

An array **arr[0..n-i]** is Bitonic if there exists an index **i**, where 0<=i<=n-1 such that

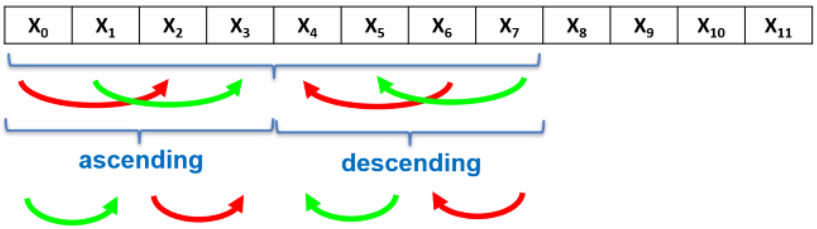
$$x_0 \le x_1 \ldots\ldots \le x_i \quad \text{and} \quad x_i \ge x_{i+1} \ldots\ldots \ge x_{n-1}$$

Example:

x_0, x_1, x_2, x_3

- sort x_0 and x_1 in **ascending** order ($\min(x_0, x_1)$, $\max(x_0, x_1)$) →
- sort x_2 and x_3 in **descending** order ($\max(x_2, x_3)$, $\min(x_2, x_3)$) ←

Then, increase the bitonic size in a power of 2 and sort with length as 2



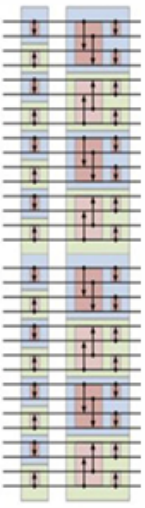
Implementation in CUDA:

```
__global__ void bitonicSort(int *data, int N) {
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockDim.x * blockIdx.x + threadIdx.x;
    extern __shared__ int s[];
    if (idx < N) s[tid] = data[idx]; // Load into shared memory (indexing per block)
    __syncthreads();
    for (unsigned int k = 2; k <= blockDim.x; k <<= 1) { // Bitonic sort algorithm
        for (unsigned int j = k >> 1; j > 0; j >>= 1) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (s[tid] > s[ixj]) {
                        int tmp = s[tid]; s[tid] = s[ixj]; s[ixj] = tmp; }
                }
                else {
                    if (s[tid] < s[ixj]) {
                        int tmp = s[tid]; s[tid] = s[ixj]; s[ixj] = tmp; }
                }
            }
            __syncthreads();
        }
    }
    if (idx < N) data[idx] = s[tid];          // Store back to global memory
}
```

Checking if ascending or descending (tid AND K)

Tid

ixj



  - **Quick Sort** (divide and conquer) = choose a pivot element; partition the array so that:
    - **elements < pivot** → go to left
    - **elements > pivot** → go to right

And then recursively apply same logic to subarrays! But how to implement it in GPU? (because quick sort involves recursion and irregular memory access)

We can use ***prefix sums*** (scan) to compute positions for elements < or > than the pivot:

- each thread checks its element against the pivot and marks it as 0 or 1
- perform an exclusive scan to compute new positions
- scatter elements into their correct positions

Quick Sort (**prefix sums-based** solution):

e.g., In_array: [9, 3, 5, 12, 4], pivot = 5

isLess = [0, 1, 0, 0, 1]

isGreater = [1, 0, 1, 1, 0]

Exclusive prefix sum on both arrays:

prefixLess = [0, 0, 1, 1, 1]

prefixGreater = [0, 1, 1, 2, 3]

numLess = prefixLess[n-1] + isLess[n-1] = 1 + 1 = 2

numGreater = prefixGreater[n-1] + isGreater[n-1] = 3 + 0

Scatter elements to output buffer:

If In_array[i] < pivot, its final position is prefixLess[i]

Else, position is numLess + prefixGreater[i]

Less: [3, 4] Greater: [9, 5, 12] => temp = [3, 4, 9, 5, 12]

Recursively sort partitions:

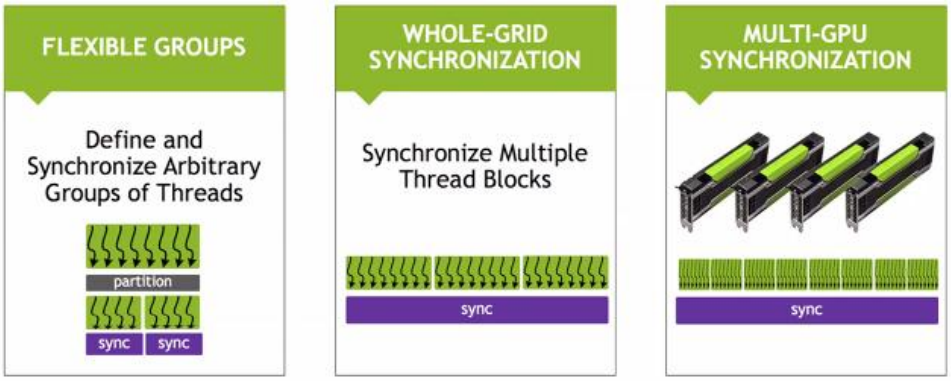
temp1 = [3, 4, 9, 5, 12] ; temp2 = [3, 4, 9, 5, 12]

Implementazione in CUDA:

```
__global__ void cdp_simple_quicksort(unsigned int *data, int left, int right, int depth){
    if (depth >= MAX_DEPTH || right - left <= INSERTION_SORT) {// If too deep or few elements left (insertion sort)
        selection_sort(data, left, right);
        return;
    }
    unsigned int *lptr  = data + left;
    unsigned int *rptr  = data + right;
    unsigned int  pivot = data[(left + right) / 2];
    while (lptr <= rptr) {            // Do the partitioning
        unsigned int lval = *lptr;    // Find the next left- and right-hand values to swap
        unsigned int rval = *rptr;
        while (lval < pivot) {    // Move the left pointer as long as the pointed element is smaller than the pivot
            lptr++;
            lval = *lptr;
        }
        while (rval > pivot) {    // Move the right pointer as long as the pointed element is larger than the pivot
            rptr--;
            rval = *rptr;
        }
        if (lptr <= rptr) {       // If the swap points are valid, do the swap!
            *lptr++ = rval;
            *rptr-- = lval;
        }
    }

    // Now the recursive part
    int nright = rptr - data;
    int nleft  = lptr - data;
    // Launch a new block to sort the left part.
    if (left < (rptr - data)) {
        cudaStream_t s;
        cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
        cdp_simple_quicksort<<<1, 1, 0, s>>>(data, left, nright, depth + 1);
        cudaStreamDestroy(s);
    }
    // Launch a new block to sort the right part.
    if ((lptr - data) < right) {
        cudaStream_t s1;
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        cdp_simple_quicksort<<<1, 1, 0, s1>>>(data, nleft, right, depth + 1);
        cudaStreamDestroy(s1);
    }
}
```

10) Cooperative Threads

Cooperative Groups in CUDA are used to extend thread flexibility. In efficient parallel algorithms, **threads cooperate and share data to perform collective computations** (so threads must be synchronized to share data, but **thread synchronization should be flexible** because the granularity of sharing differs from algorithm to algorithm)



In *CUDA 9*, **Cooperative Groups** are added (a flexible model for synchronization and communication within groups of threads) [*obtain more flexibility on threads in a group*]

To synchronize cooperative threads, we can use **`__syncthreads()`** [a barrier across all threads in a block], but how we can manage synchronization among cooperative groups?

So new APIs (both CUDA and host-side) needed to be defined to manage synchronization in groups of threads

⚠ To use cooperative groups, we need GPU compatible with at least CUDA 9, with compute capability 3.0+ and including **`#include <cooperative_groups.h>`**

Threads groups (**`thread_group`**) = fundamental type in cooperative groups; it's a handle to a group of threads; some methods are:

- `unsigned` **`size()`** → size of a thread group
- `unsigned` **`thread_rank()`** → index of the calling thread in the group
- `bool` **`is_valid()`** → validity of a group

Collectives (collective operations) = operations for the synchronization or communication among a specified set of threads. We can use **`cooperative_groups::sync()`** (or **`g.sync()`**) to synchronize all threads in the group, as a barrier for all threads.
Esempio (*parallel reduction with cooperative groups*):

```
using namespace cooperative_groups;
__device__ int reduce_sum(thread_group g, int *temp, int val)
{
    int lane = g.thread_rank();
    // Each iteration halves the number of active threads
    // Each thread adds its partial sum[i] to sum[lane+i]
    for (int i = g.size() / 2; i > 0; i /= 2)
    {
        temp[lane] = val;
        g.sync();     // wait for all threads to store
        if(lane<i) val = val + temp[lane + i];
        g.sync();     // wait for all threads to load
    }
    return val;       // note: only thread 0 will return full sum
}
```

Thread blocks (**`thread_block`**) = threads in 1 block (key feature is that we can arbitrary size for thread blocks); it's a handle to the group of threads in a CUDA thread block, initialized as:

```
thread_block block = this_thread_block();
    auto block = this_thread_block();
```

Every thread that executes that line has its instance of the variable block. How to sync a thread block →

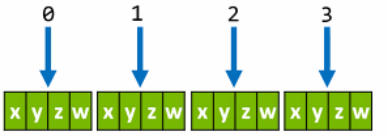
```
__syncthreads();
block.sync();
cg::synchronize(block);
this_thread_block().sync();
cg::synchronize(this_thread_block());
```

Esempio:

Thread blocks

e.g., compute the sum of all values in an input array

```
__device__ int thread_sum(int *input, int n) {
    int sum = 0;
    for(int i = blockIdx.x*blockDim.x+threadIdx.x; i<n/4; i+=blockDim.x*gridDim.x)
    {
        int4 in = ((int4*)input)[i];
        sum += in.x + in.y + in.z + in.w;
    }
    return sum;
}
__global__ void sum_kernel_block(int *sum, int *input, int n){
    int my_sum = thread_sum(input, n);
    extern __shared__ int temp[];
    auto g = this_thread_block();
    int block_sum = reduce_sum(g, temp, my_sum);
    if (g.thread_rank() == 0) atomicAdd(sum, block_sum);
}
```



grid-stride loop

It allows any number of threads to process an array of any size

Reduction per thread

Grid-stride vs Block-stride *loops*: patterns that let threads process more data than the # of threads submitted, by having each thread jump forward by a fixed "stride" each iteration (stride might be different) [stride = total number of threads in the entire grid] [es. in Block-stride loop, each block processes 1 chunk of the image, while the grid processes 3 image chunks]

Partitioning groups = parts or associations of threads from existing groups: flexible method to create new groups, enables cooperation and sync at a finer granularity; **`cg::tiled_partition(block, 32)`** partitions a thread block into tiles of 32 threads (so partitioning allows better granularity and speedup)

Modularity = main power and utility of cooperative groups; allows the correct operation of a function with a consistent interface for thread groups of a variety of sizes; prevents to cause race conditions and deadlock by avoiding invalid assumptions about which threads will call a function concurrently.

Esempio:

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition(this_thread_block(), 32);
reduce(g, ptr, myVal);
```

```
__device__ int reduce(thread_group g, int *x, int val) {
  int lane = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    x[lane] = val;        g.sync();
    if (lane < i) val += x[lane + i];  g.sync();
  }
  return val;
}
```

⚠ Optimization of GPU warp size: cooperative groups provides an alternative version of **`tiled_partition()`** that takes the tile size as a *template parameter*, returning a statically sized group called a **`thread_block_tile`** (knowing tile size at compile time allows > optimization (*loop unrolling*))

e.g., sum reduction with statically sized thread_block_tile

```
template <typename group_t>
__device__ int reduce_sum(group_t g, int *temp, int val){
    int lane = g.thread_rank();
    // Each iteration halves the number of active threads
    // Each thread adds its partial sum[i] to sum[lane+i]
    #pragma unroll
    for (int i = g.size() / 2; i > 0; i /= 2){
        temp[lane] = val;
        g.sync(); // wait for all threads to store
        if (lane < i) val += temp[lane + i];
        g.sync(); // wait for all threads to load
    }

    return val; // note: only thread 0 will return full sum
}
```

When **tile size** and **warp size** are the same, it is possible to avoid sychnonizations

Always explicitly synchronize the thread groups (implicitly synchronized programs have race conditions)

Warp-level collectives → thread block tiles provide an **API for warp-level collective functions**:

- **`.shfl()`** → shifts the value from a line across other lines in a warp: allows a thread to read a value directly from another thread's register within the same warp, without using shared memory

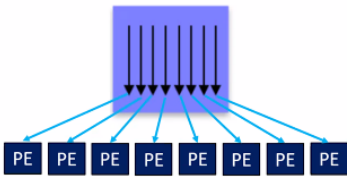
```
int result = __shfl_sync(0xffffffff, val, 0);
val = Tile32.shfl(val, 0);
```

Lane ID

Directly reads a value from a given Lane ID

Example with a warp of 8 threads, **value** = lane ID

Lane: 0 1 2 3 4 5 6 7
Value: 8 1 2 3 4 5 6 7
Result: 8 8 8 8 8 8 8 8



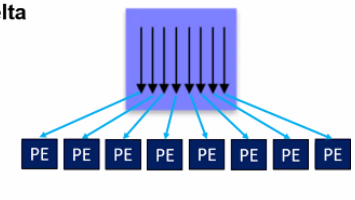
- **`.shfl_down()`** → shifts the value downward across the warp

```
value_from_thread = __shfl_down_sync(0xffffffff, value, delta);
val = Tile32.shfl_down(val, delta);
```

The thread gets the value from the thread whose lane ID is **lane + delta**

Example with a warp of 8 threads, **value** = lane ID, **delta** = 2:

Lane: 0 1 2 3 4 5 6 7
Value: 0 1 2 3 4 5 6 7
Result: 2 3 4 5 6 7 ? ?



- **`.shfl_xor()`** → exchanges values based on XOR of lane IDs (useful for binary-tree patterns like warp-reductions and scans)

Lane ID: 0 1 2 3 4 5 6 7
Value: 10 11 12 13 14 15 16 17

```
new_val = Tile32.shfl_xor(val, 1);
```

Pair lines: 0↔1, 2↔3, 4↔5, 6↔7 …

lane 0 → 0 ^ 1 = 1 → gets value 11
lane 1 → 1 ^ 1 = 0 → gets value 10
lane 2 → 2 ^ 1 = 3 → gets value 13
lane 3 → 3 ^ 1 = 2 → gets value 12
lane 4 → 4 ^ 1 = 5 → gets value 15
lane 5 → 5 ^ 1 = 4 → gets value 14
lane 6 → 6 ^ 1 = 7 → gets value 17
lane 7 → 7 ^ 1 = 6 → gets value 16

- **`.any()`** → warp-level voting function that evaluates predicate for all threads in a warp (in questo modo si evita la divergence; funziona come una normale any, quindi basta che almeno 1 thread abbia il predicato != 0)

```
int result = __any_sync(0xFFFFFFFF, pred);
bool any_true = Tile32.any(predicate);
```

- **`.ballot()`** → come la any, ma ritorna bitmask con 1 bit per thread del gruppo; ogni bit della mask viene messo a 1 se il predicato è vero per quel thread

```
Unsigned int result = __ballot_sync(mask, predicate);
unsigned mask = tile32.ballot(predicate);
```

- **`.match_any()`** → invece di dire "chi ha cond == true?" (ballot), mask dice "chi ha il mio stesso valore?"
- **`.match_all()`** → torna true se ogni thread nel warp/tile ha lo stesso valore

Thread concurrency: nei kernel spesso si hanno situazioni di **data-dependent divergence** (come nell'img); l'SM disabilita i threads che non entrano nel branch, mentre i thread che rimangono active nel warp (*coalesced threads*) possono essere raggruppati con **`coalesced_threads()`** [ricorda sempre solo nel warp, non si raggruppano inter-warp]:

```
__global__ void thresholdKernel(float *in, float *out, float t) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    …
    float x = in[idx];
    if (x > t) {
        out[idx] = x * 2.0f;
    } else {
        out[idx] = 0.0f;
    }
    …
}
```

```
auto block = this_thread_block();
if (block.thread_rank() % 2) {
    coalesced_group active = coalesced_threads();
    ...
    active.sync();
}
```

The coalesced threads can be synchronized through **`active.sync()`**

Grid synchronization:

A set of threads within the same grid, **guaranteed to be resident on the device**

- Device must support the **cooperativeLaunch** property

To synchronize across the grid, from within a kernel, you would simply use the **grid.sync()** function

- Use the **cudaLaunchCooperativeKernel()** instead of **<<<…,..>>>()**
- **Dynamic kernel** submission **not supported**

```
cg::grid_group grid = cg::this_grid();
grid.sync();
```

⚠ Constraints of cooperative groups: **all blocks must run concurrently**; it might limit the gridDim based on the GPU's SMs and resources (if grid is too large, launch of the kernel in cooperative groups will fail). A queue strategy can be applied to execute a max number of blocks to resolve this

⚠ ***Multi-grids*** → can be used to synchronize more GPUs devices together (in situazioni di multi-device)

ROOFLINE MODEL (measuring and improving the performance in parallel programs for GPUs)

Identify a highly used metric to analyze performance of GPU. Motivazione per trovare un modo per capire se il nostro "programma" è buono è l'**hetereogeneous** computing program **flow model**, dove avvengono trasferimenti [**Q**] tra slow memory (global) e fast memory (dove avvengono le operazioni [**W**]); definiamo:

Intensity = W/Q = n° arithmetic operations / n° main memory operations

[**computational intensity** = ratio of # of computation performed to the # of data movements; quindi meglio avere high computation intensity, mentre se la abbiamo low ci sono tanti ***bottlenecks***]

Example: **Intensity** = W/Q (e.g., flops per byte)

Calculate the computational complexity of the addition of two vectors (VADD) with 10 single-floating elements:

C[i]=A[i]+B[i]; for i=1,2,...,n

Total operations = n:10

W

Total data movement: 2n + n => 3n: 30 elements.

A[] B[] C[]

Each element is 4 bytes (32-bit float)

The total data transfer in bytes is: 30×4 = 120 bytes. Q

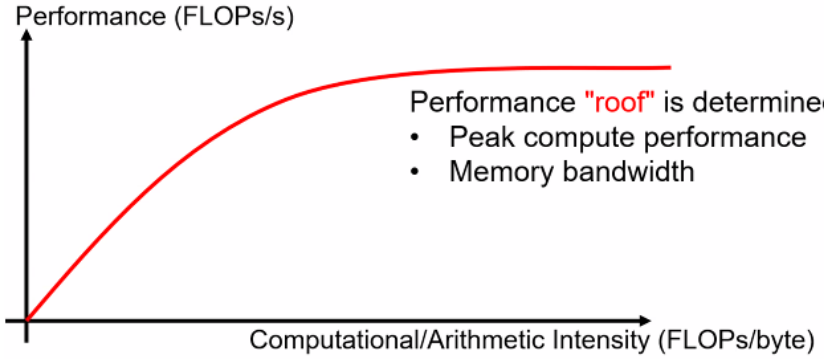
Intensity = W/Q

Intensity = 10/120

Intensity = 0.0833

⚠ Questi concetti sono strettamente legati allo speedup (Amdahl e Gustafson)!

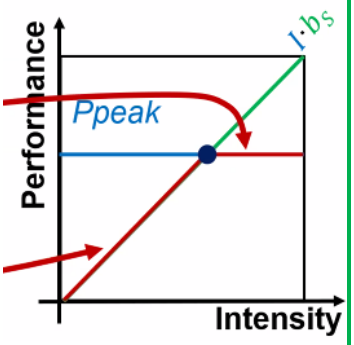
So, **the solution adopted is the ROOFLINE MODEL**: a intuitive visual performance model, that estimates the performance of kernels running on different configurations (multi-core, GPUs...) and shows the hw limitations (and the benefit for optimization purposes). Le **2 metriche** usate in questo modello sono i 2 assi del grafico, ovvero:



It's an **optimistic model with simplistic view of hw and sw**; the app bottleneck in *P* is either:

- the work execution → $\boldsymbol{P_{peak}}$ [flop/s]
- the data path → $\boldsymbol{I * b_s}$ [flop/byte * byte/s]

The **naïve roofline model** says $\boldsymbol{P = \min(P_{peak}, I * b_s)}$; if *high intensity*, *P* is limited by execution, while if *low intensity*, *P* is limited by data transfer. The transition point is the intersection point between high intensity and low intensity (best use of resources), so that $\boldsymbol{P_{peak} = I * b_s}$



Prerequisites for roofline model:

- data transfer and core execution overlap perfectly
- slowest limiting factor "wins" (all others are assumed to have no impact)
- data access latency is ignored (achievable bandwidth is the limit)
- device must be able to saturate the bandwidth bottleneck (in fact, always model the full device)

The **refined/extended roofline model** instead says this:

*P*max= Applicable peak performance, assuming that data comes from the level 1 cache (this is not necessarily *P*peak)

e.g., *P*max= 176 GFlop/s

I = Computational intensity ("work" per byte transferred) over the slowest data path utilized

e.g., I = 0.167 Flop/Byte code balance $BC = 1/I$

B_S = Applicable (saturated) peak bandwidth of the slowest data path utilized

e.g., B_S= 56 GByte /s [Byte/s] [Byte/Flop]

Performance limit:

$$P = \min(P\max, I \cdot b_S) = \min(P\max, b_S/BC)$$

⚠ Si può calcolare a mano il transition point del nostro programma usando il modello appena descritto, ma in realtà **se si usano i profiler questi lo fanno già per noi sulla base del nostro codice**!

How to build roofline model in reality?

How to Build the Roofline

1. Measure GPU's peak performance (FLOPs). (manual)
2. Measure memory bandwidth (GB/s). (manual)
3. Plot roofs based on these measurements.
4. Determine the Computational intensity of a target application (Kernel)
   - **Tools:** Use profilers (e.g., NVIDIA Nsight).

1. Place workload points based on I and performance

Example (determine intensity and roofline for a dense GEMM application [N= 1024, double precision]):

GPU specs: **Peak:** 15.0 TFLOP/s **Bandwidth:** 900 GB/s

Ridge point = 15e12 / 900e9 = **16.666667 FLOP/byte**

Dense GEMM:*(Square GEMM (N×N × N×N) FLOPs) → multiply and add (2 FLOPs)*

FLOPs = 2 · N^3 = 2 · 1024^3 = **2,147,483,648 FLOP**

Bytes (ideal minimal) = (# operands · size per operand · operand bytes)
= 3 · N^2 · 8 = 3 · 1024^2 · 8 = **25,165,824 bytes**

Operational intensity (OI) = 2,147,483,648 / 25,165,824 = **85.333333 FLOP/byte**

Achievable perf = min(peak, OI·BW) = min(15 TFLOP/s, 85.3333 · 900 GB/s)
= **15.000 TFLOP/s → compute-bound**

Sotfware libraries for parallel programming with CUDA

Oggi ci sono strategie e GPU che permettono una potenza computazionale assurda, ma usare in modo efficiente queste risorse è complicato (scrivere custom GPU code per single operazioni richiede conoscenza ottima delle GPU, dell'architettura, delle gerarchie di memoria e della parallelization). Using **GPU libraries** is essential (highly optimized and tuned for GPU hw):

- **Pro**: **performance**, **productivity** (il focus è sullo sviluppo degli algoritmi, non sulla low-level GPU optimization, perchè se ne occupa già la libreria), **portability** + **maintainability**, **reliability** + **accuracy**, **ecosystem integration** (molte librerie sono integrate in frameworks/tools, es. PyTorch)
- **Contro**: **limited flexibility**, **memory constaints** (libraries can require *temporary workspace* or *buffers* for operations), "***black-box***" **nature**, **integration overhead**

STANDARD LIBRARIES are **CUDA math**, **cuBLAS**, **cuFFT**, **cuSPARSE**, **cuSOLVER**, **cuRAND**, **nvGRAPH**, **Thrust**, **NPP**… (guarda slides se vuoi maggiori dettagli su come funzionano e su che funzioni offrono)