HPC Workshop

NATIONAL TECHNICAL UNIVERSITY OF ATHENS

EuroCC@Greece

Large-Scale Scientific Computations

School of Chemical Engineering

Computer Center

School of Mechanical Engineering

Parallel CFD and Optimization Unit

School of Electrical and Computer Engineering

Computing Systems Laboratory

Introduction to modern graphics processing units (GPU) architecture and programming in CUDA

Xenofon Trompoukis, Dr Mechanical engineer

School of Mechanical Engineering, NTUA,
Parallel CFD & Optimization Unit
email: xeftro@gmail.com

Why GPUs ?

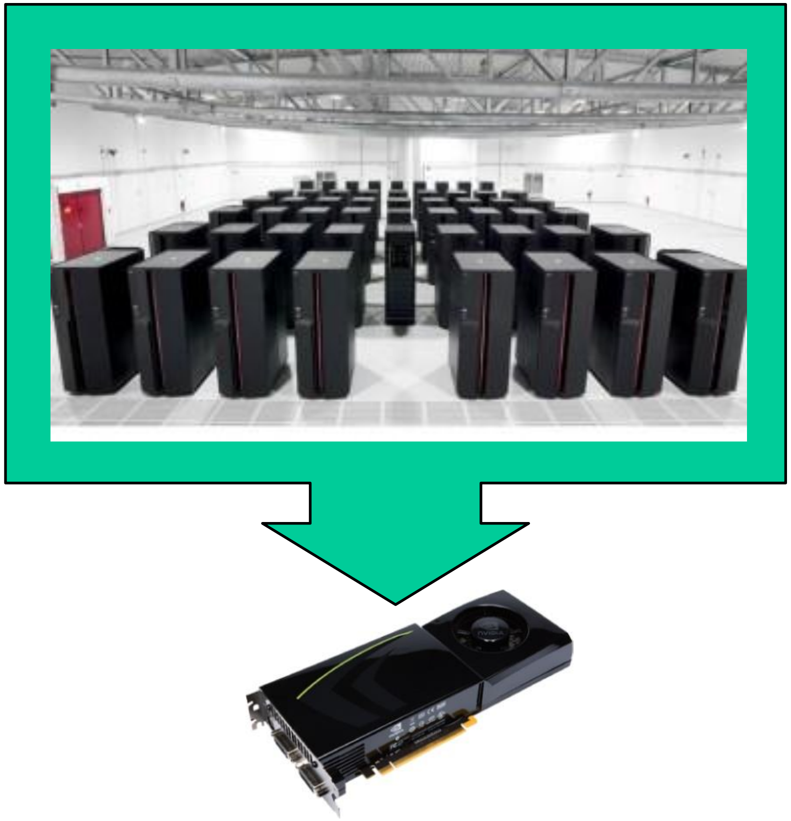
❑ Parallel processing units

❑ High floating-point operations rate (double and single precision arithmetic)

❑ GPU embedded, low latency, RAM

❑ Various programming environments

❑ Low cost & energy consumption based on their computational power



Why GPUs ?

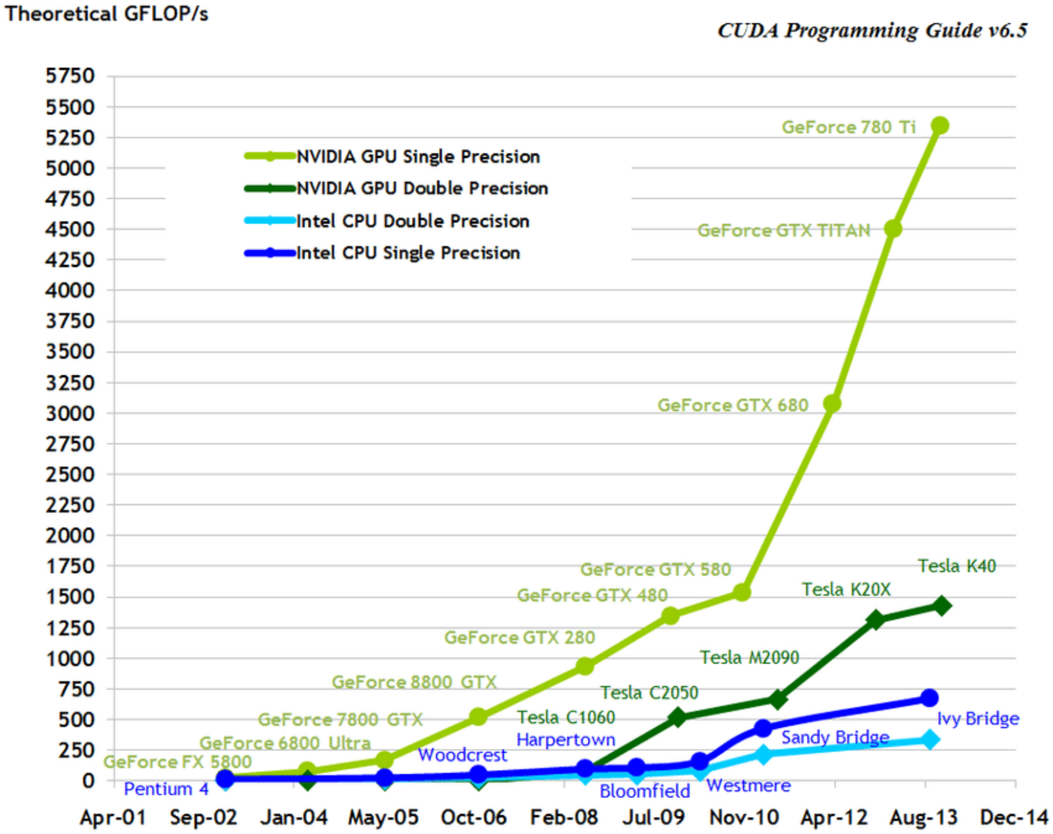
❑ Parallel processing units

❑ High floating-point operations rate (double and single precision arithmetic)

❑ GPU embedded, low latency, RAM

❑ Various programming environments

❑ Low cost & energy consumption based on their computational power



CUDA Programming Guide v6.5

Why GPUs ?

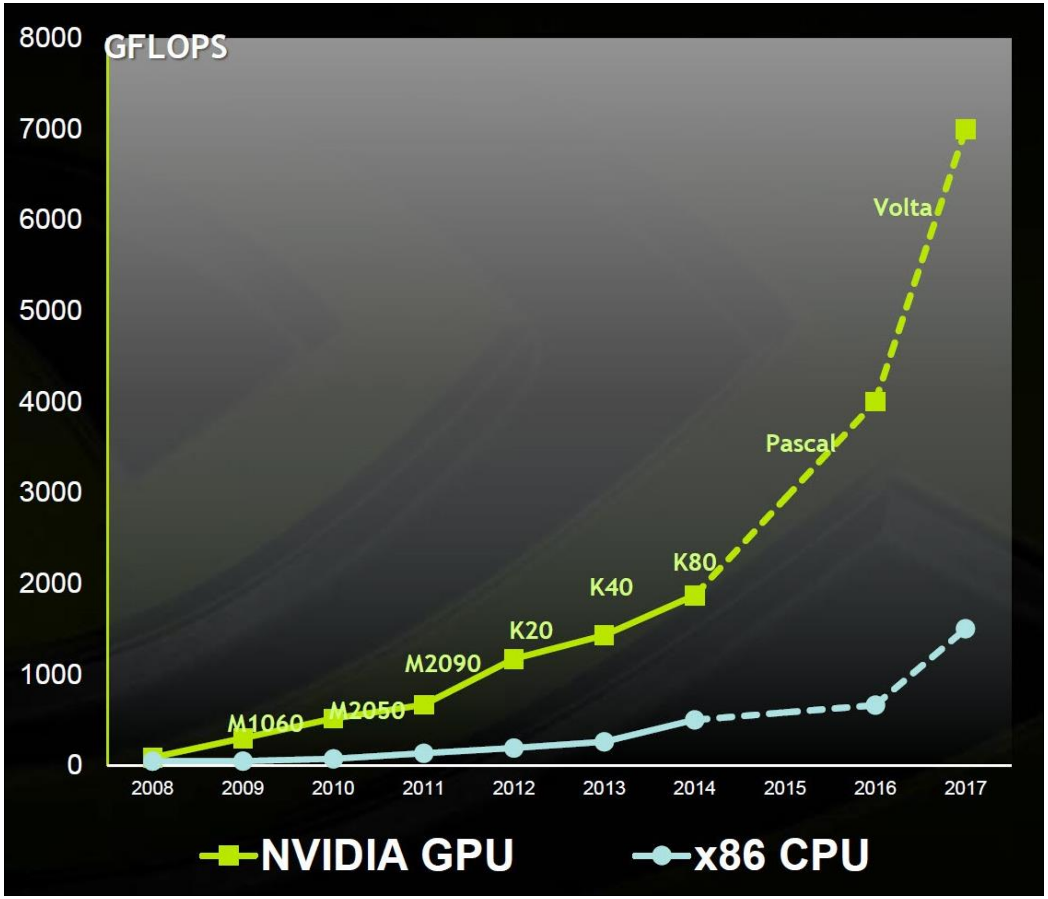
❑ Parallel processing units

❑ High floating-point operations rate (double and single precision arithmetic)

❑ GPU embedded, low latency, RAM

❑ Various programming environments

❑ Low cost & energy consumption based on their computational power



Why GPUs ?

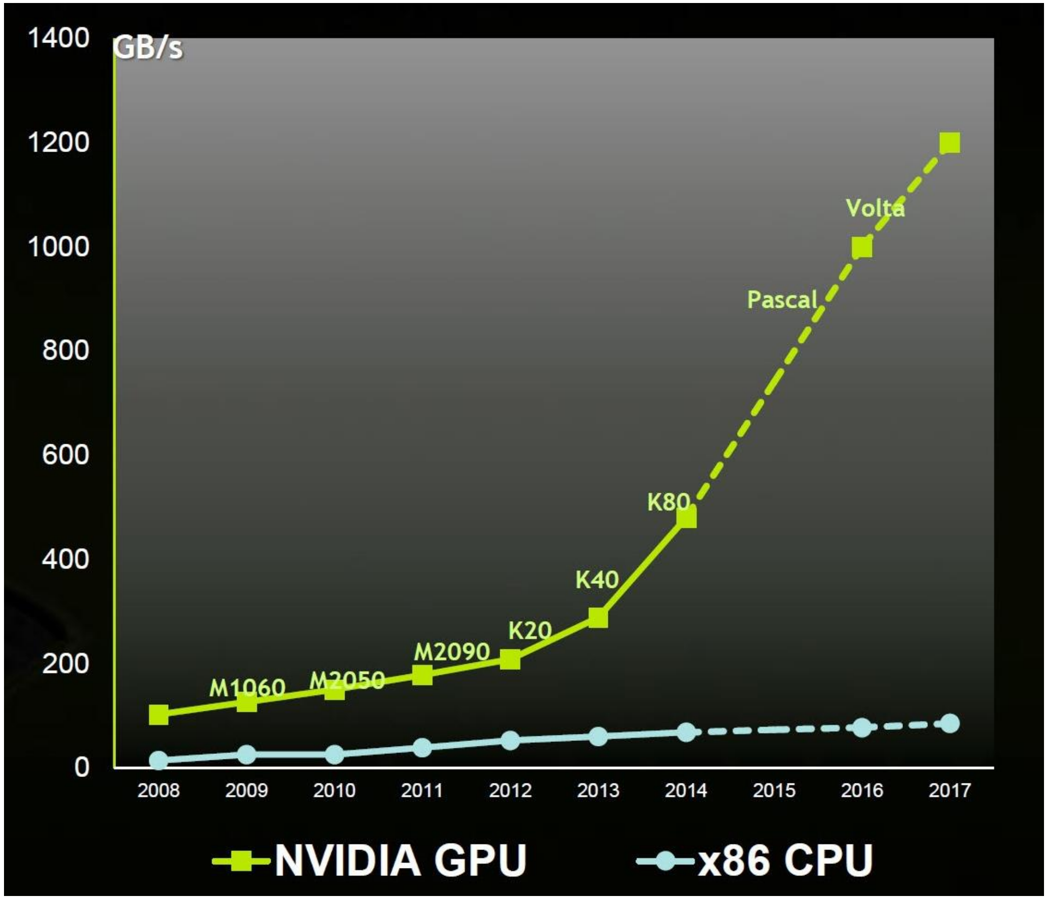
❑ Parallel processing units

❑ High floating-point operations rate (double and single precision arithmetic)

❑ GPU embedded, low latency, RAM

❑ Various programming environments

❑ Low cost & energy consumption based on their computational power



Why GPUs ?

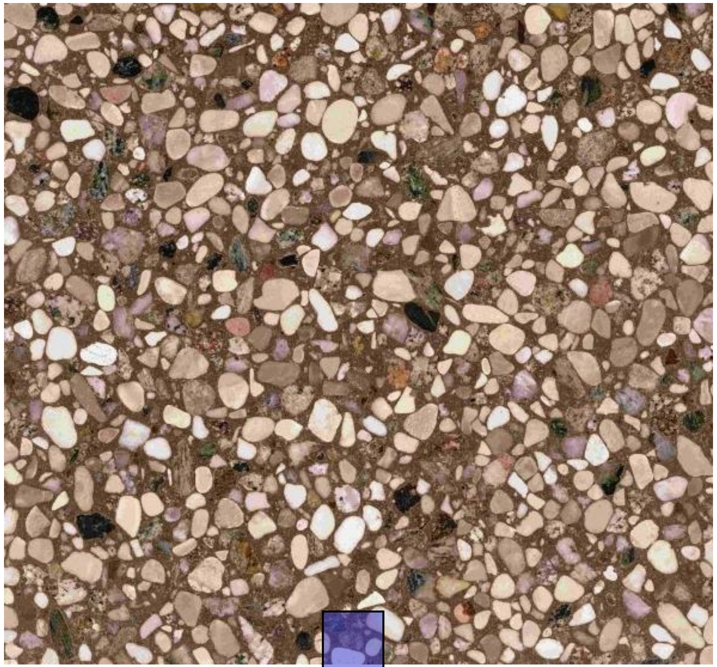
❑ Parallel processing units

❑ High floating-point operations rate (double and single precision arithmetic)

❑ GPU embedded, low latency, RAM

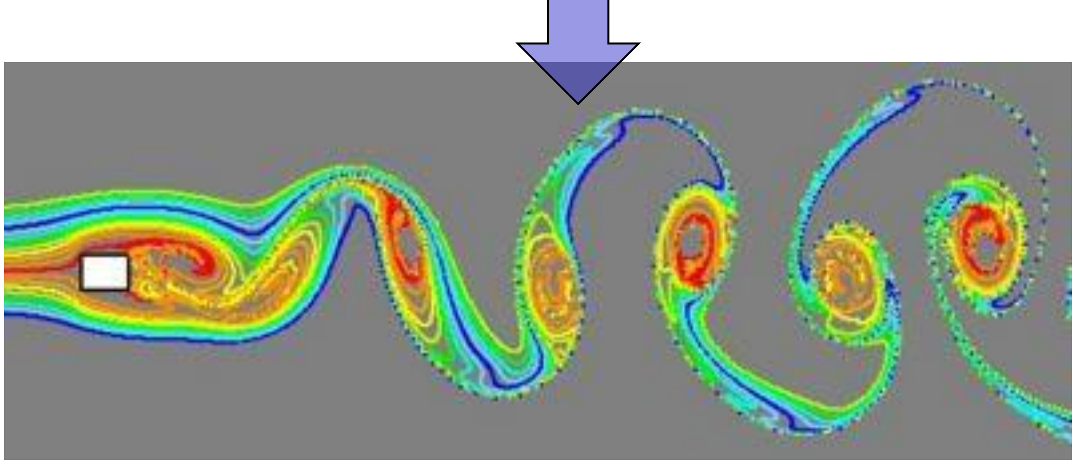
❑ Various programming environments

❑ Low cost & energy consumption based on their computational power



OpenCL: Cross platform implementation
- C++

CUDA: Developed by NVIDIA, specialized for NVIDIA GPUs
- C++
- FORTRAN
- Python



Why GPUs ?

❑ Parallel processing units

❑ High floating-point operations rate (double and single precision arithmetic)

❑ GPU embedded, low latency, RAM

❑ Various programming environments

❑ Low cost & energy consumption based on their computing power

Why GPUs ?

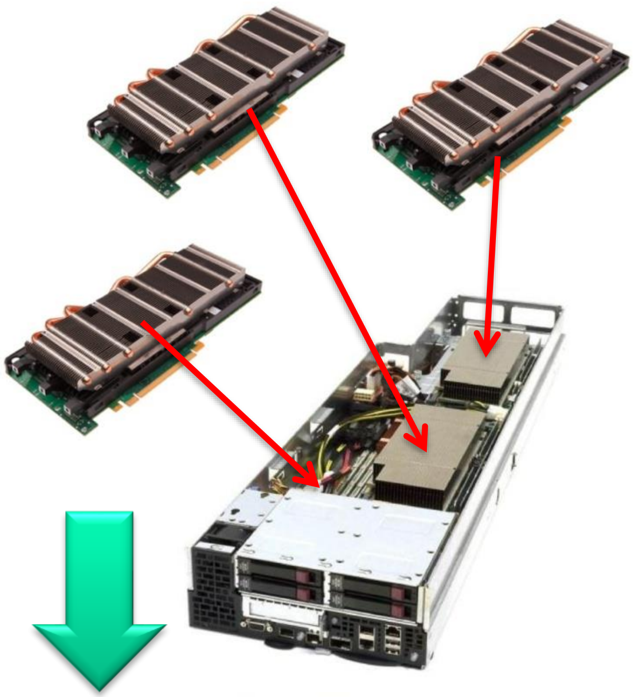
❑ Parallel processing units

❑ High floating-point operations rate (double and single precision arithmetic)

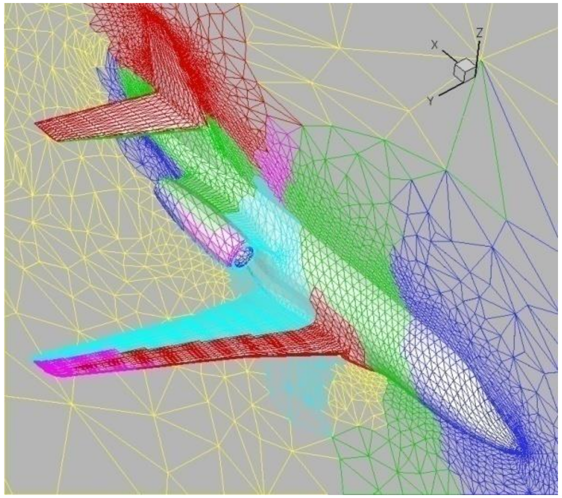
❑ GPU embedded, low latency, RAM

❑ Various programming environments

❑ Low cost & energy consumption based on their computational power



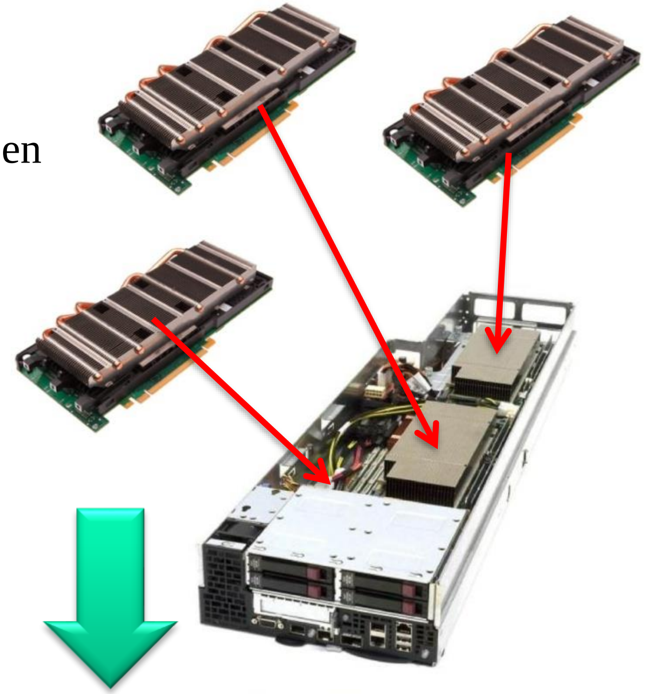
Why GPUs ?



❑ 1 sub-domain per GPU.

❑ MPI → data transactions between GPUs on different computational nodes.

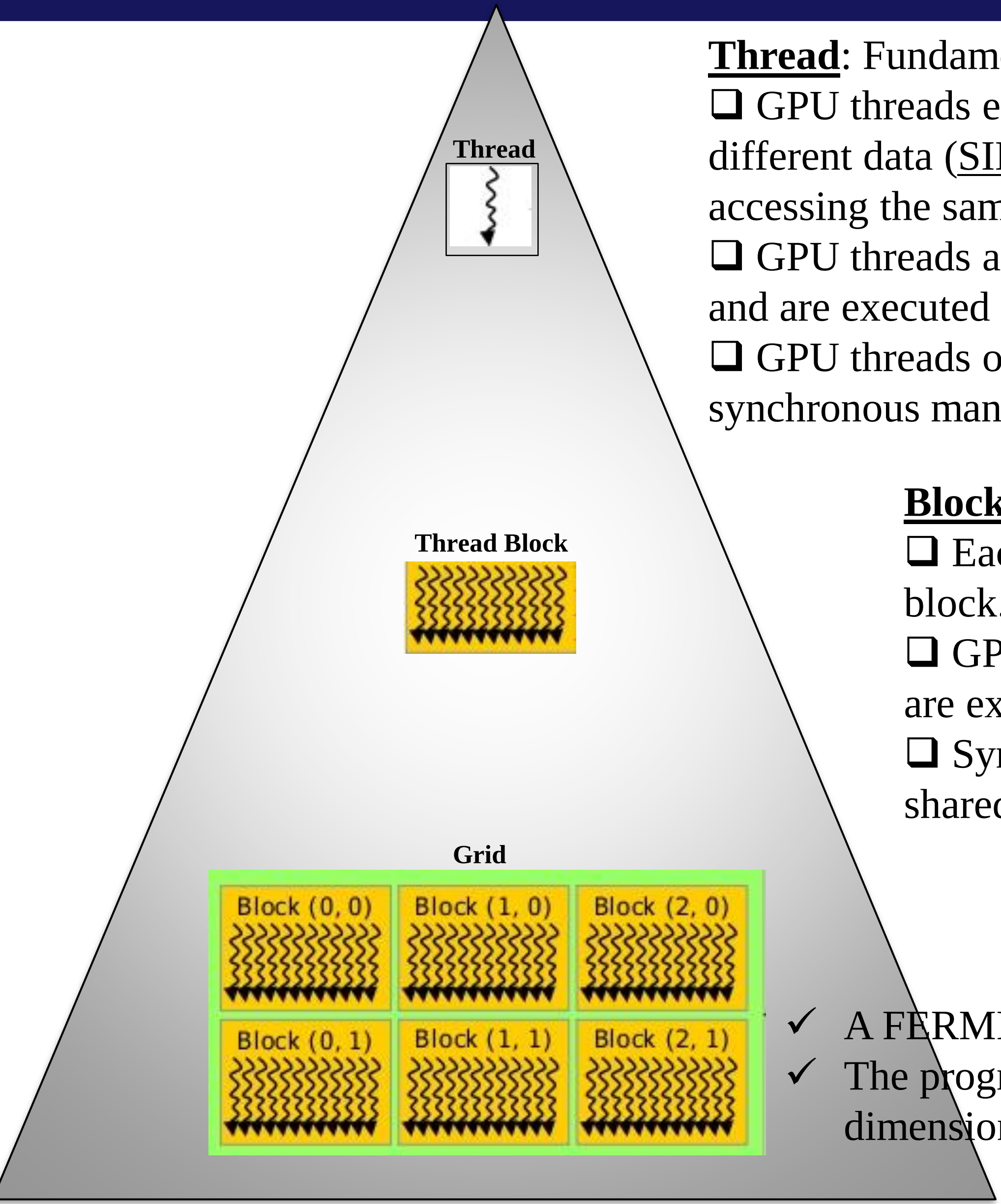
❑ Shared “host” memory → data transactions between GPUs on the same node.



Why GPUs ?

GPUs = Powerful, massively parallel CPU co-processors

GPU Architecture



Thread: Fundamental computational unit

□ GPU threads execute the same fragment of code (kernel) using different data (SIMT: Single Instruction Multiple Threads) accessing the same (device) memory.

□ GPU threads are grouped in warps (i.e. group of 32 threads) and are executed at GPU multiprocessors.

□ GPU threads of the same warp are executed in parallel in a synchronous manner.

Block: Cluster of warps

□ Each multiprocessor can execute at least a thread block.

□ GPU block threads, which belong to different warps, are executed in parallel and asynchronous manner.

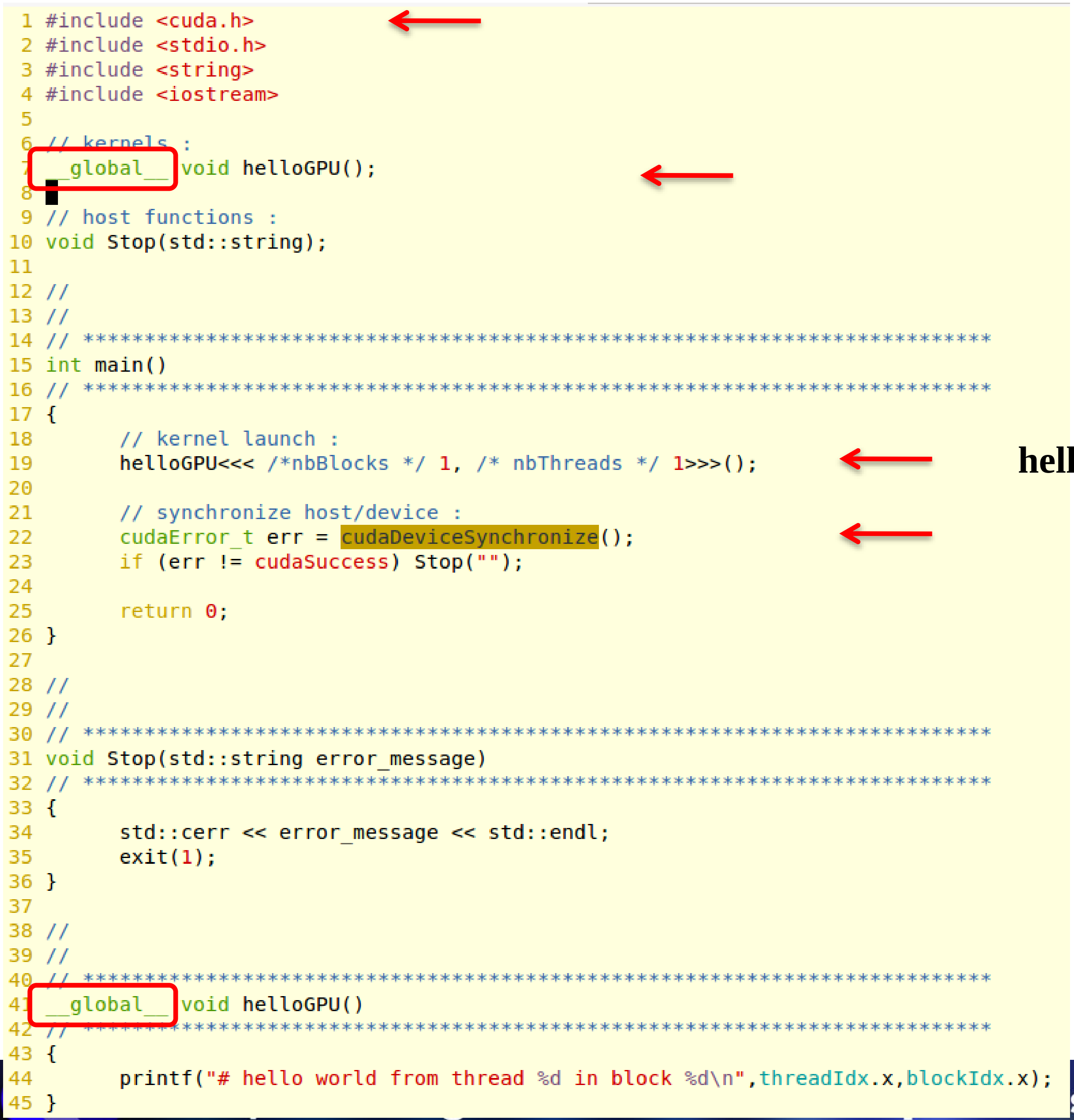
□ Synchronization and fast data transactions through shared memory

Grid: Cluster of thread blocks

✓ A FERMI GPU can execute up to 24,576 threads in parallel

✓ The programmer defines the thread block and grid dimensions

Hello World



```
#include <cuda.h>
#include <stdio.h>
#include <string>
#include <iostream>

// kernels :
__global__ void helloGPU();

// host functions :
void Stop(std::string);

//
//
// **************************************************************************
int main()
// **************************************************************************
{
        // kernel launch :
        helloGPU<<< /*nbBlocks */ 1, /* nbThreads */ 1>>>();

        // synchronize host/device :
        cudaError_t err = cudaDeviceSynchronize();
        if (err != cudaSuccess) Stop("");

        return 0;
}

//
//
// **************************************************************************
void Stop(std::string error_message)
// **************************************************************************
{
        std::cerr << error_message << std::endl;
        exit(1);
}

//
//
// **************************************************************************
__global__ void helloGPU()
// **************************************************************************
{
        printf("# hello world from thread %d in block %d\n",threadIdx.x,blockIdx.x);
}
```

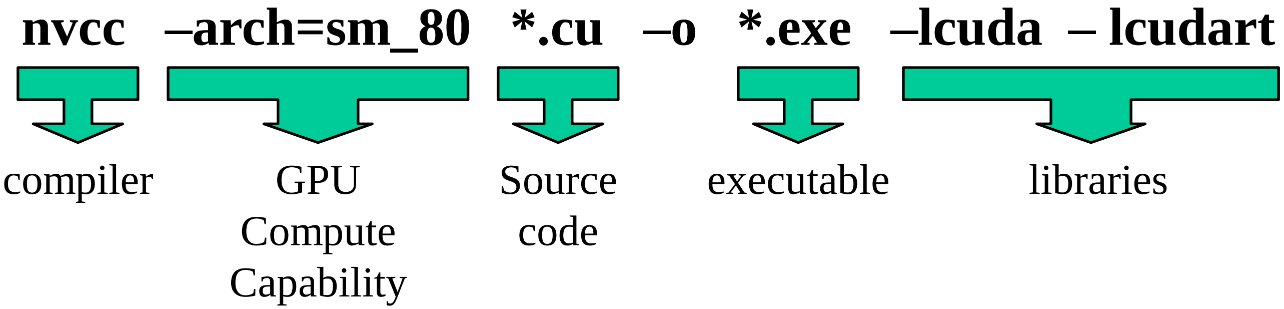
#include <cuda.h>

__global__ void helloGPU();

helloGPU<<<GridDim, BlockDim>>>();

Host-Device Synchronization

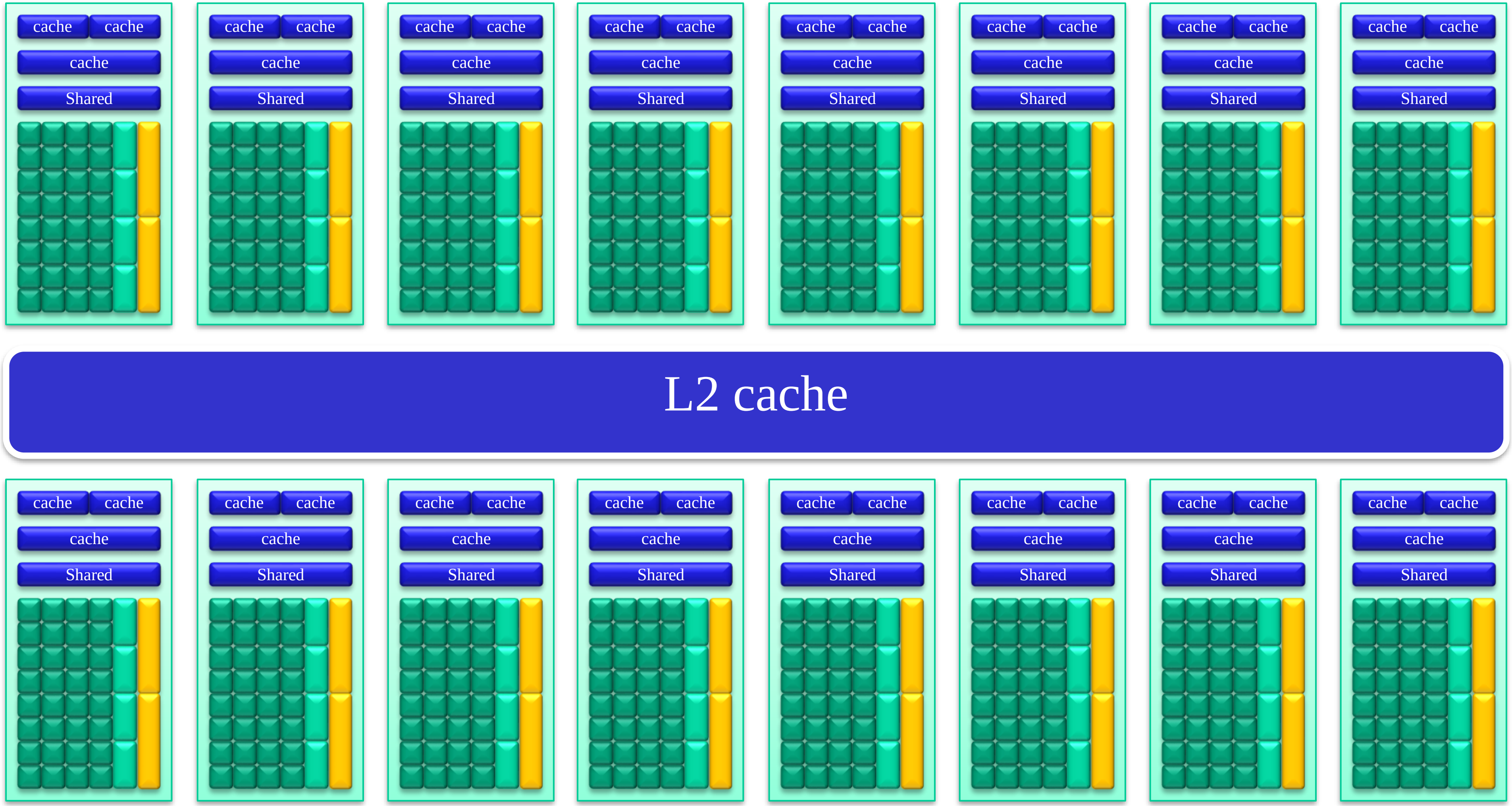
compile



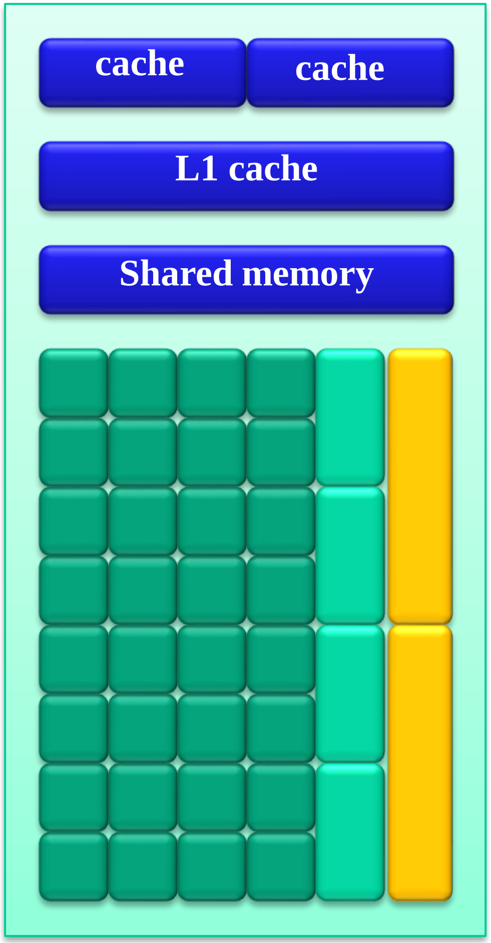
nvcc –arch=sm_80 *.cu –o *.exe –lcuda – lcudart

GPU Architecture

Fermi:16 multiprocessors



GPU Architecture



1 FERMI multiprocessor consists of:

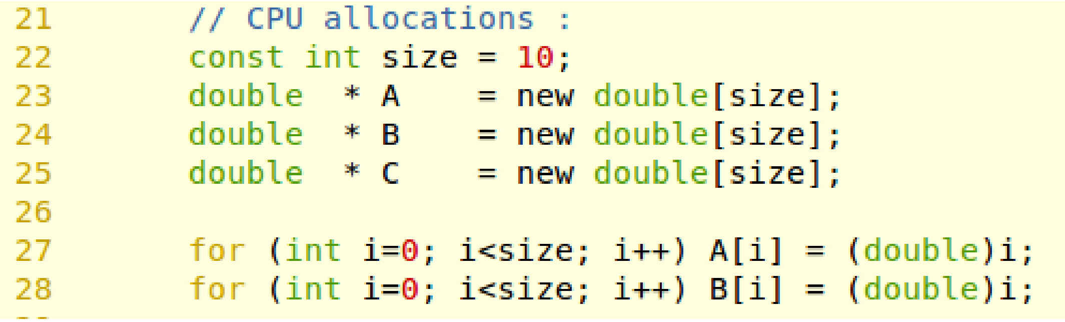
- ❑ 32 (CUDA) cores
- ❑ 4 Special Function Units (SFUs)
- ❑ 2 warp schedulers
- ❑ Shared memory
- ❑ cache memory (L1, constant & texture)
- ❑ 32768 32-bit registers

GPU Architecture

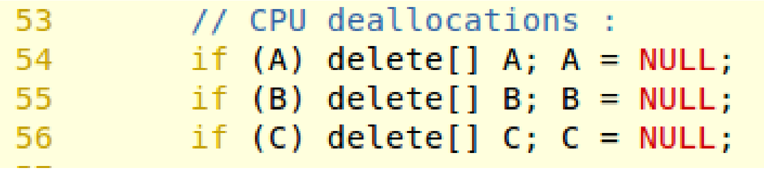
❑ Thread blocks are “split” into the multiprocessors based on kernel’s requirements on registers and shared memory. Then, the warp schedulers of each multiprocessor organize block threads into warps.

❑ The best performing block size is related only with the GPU architecture and kernel requirements not with the application itself.

Vector summation

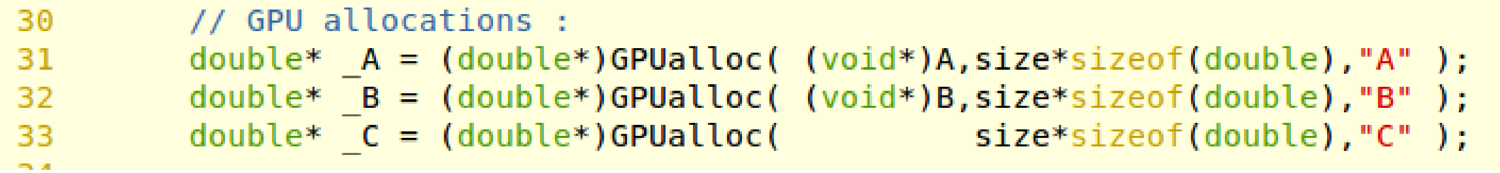


```
        // CPU allocations :
        const int size = 10;
        double  * A     = new double[size];
        double  * B     = new double[size];
        double  * C     = new double[size];

        for (int i=0; i<size; i++) A[i] = (double)i;
        for (int i=0; i<size; i++) B[i] = (double)i;
```

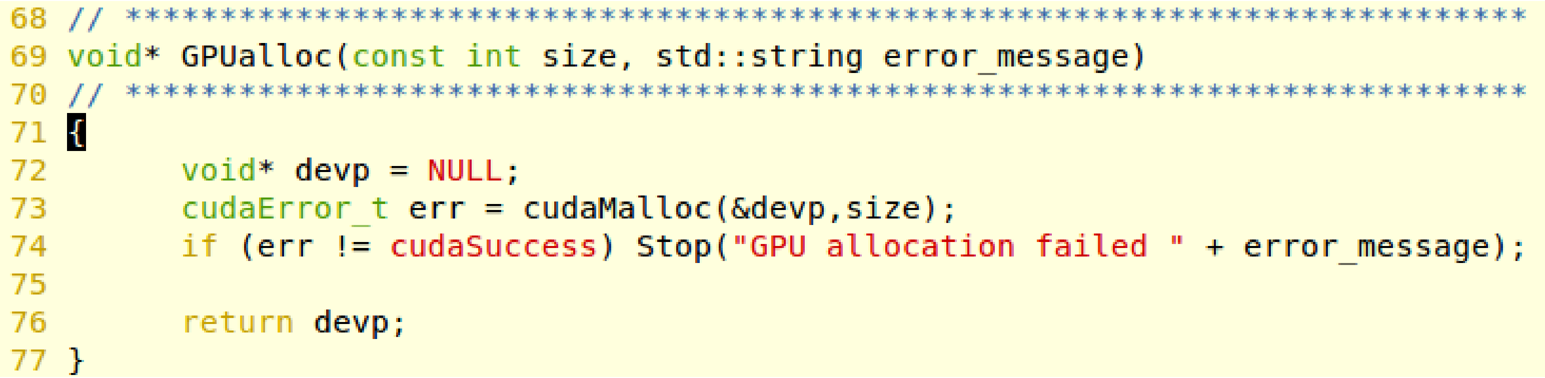


```
        // CPU deallocations :
        if (A) delete[] A; A = NULL;
        if (B) delete[] B; B = NULL;
        if (C) delete[] C; C = NULL;
```

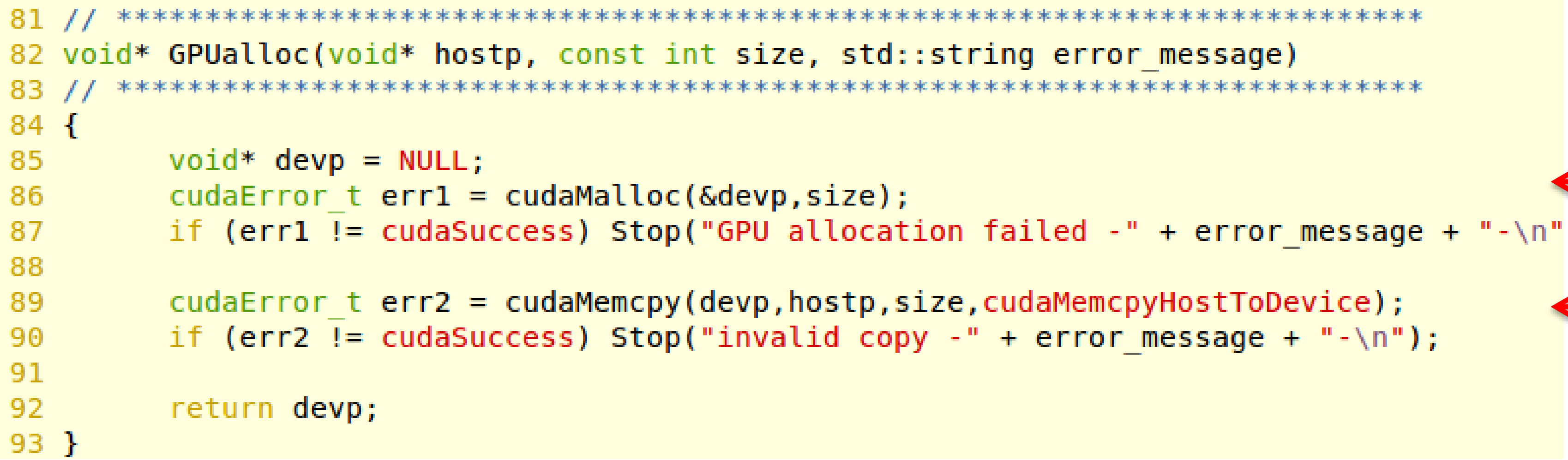
Vector summation



```
      // GPU allocations :
      double* _A = (double*)GPUalloc( (void*)A,size*sizeof(double),"A" );
      double* _B = (double*)GPUalloc( (void*)B,size*sizeof(double),"B" );
      double* _C = (double*)GPUalloc(          size*sizeof(double),"C" );
```

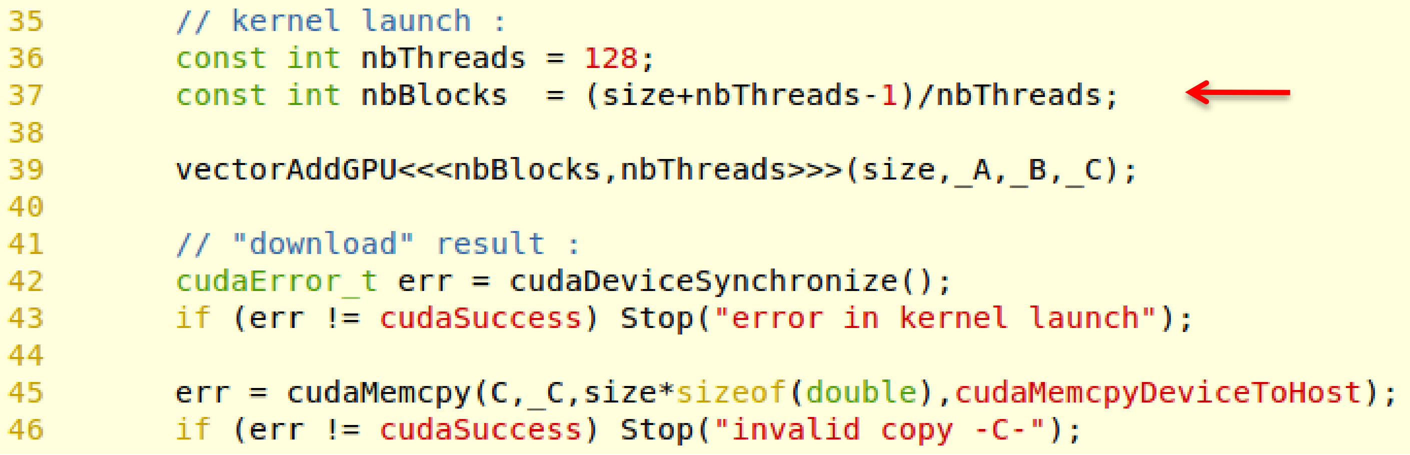


```
// ************************************************************************
void* GPUalloc(const int size, std::string error_message)
// ************************************************************************
{
      void* devp = NULL;
      cudaError_t err = cudaMalloc(&devp,size);
      if (err != cudaSuccess) Stop("GPU allocation failed " + error_message);

      return devp;
}
```

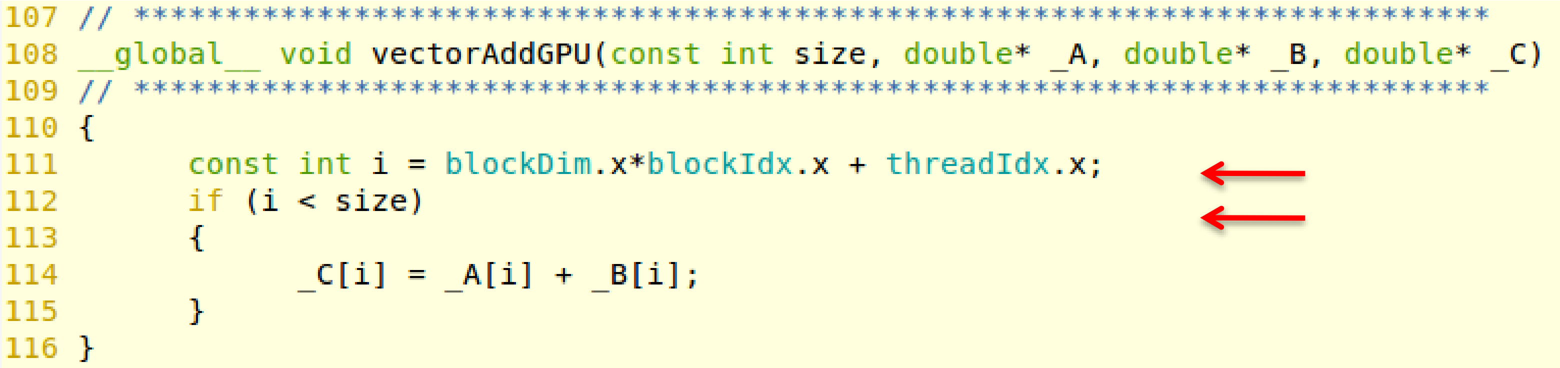


```
// ************************************************************************
void* GPUalloc(void* hostp, const int size, std::string error_message)
// ************************************************************************
{
      void* devp = NULL;
      cudaError_t err1 = cudaMalloc(&devp,size);
      if (err1 != cudaSuccess) Stop("GPU allocation failed -" + error_message + "-\n"

      cudaError_t err2 = cudaMemcpy(devp,hostp,size,cudaMemcpyHostToDevice);
      if (err2 != cudaSuccess) Stop("invalid copy -" + error_message + "-\n");

      return devp;
}
```

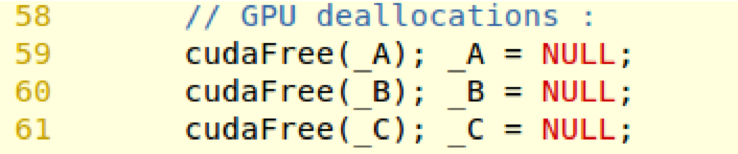
Vector summation



```
        // kernel launch :
        const int nbThreads = 128;
        const int nbBlocks  = (size+nbThreads-1)/nbThreads;

        vectorAddGPU<<<nbBlocks,nbThreads>>>(size,_A,_B,_C);

        // "download" result :
        cudaError_t err = cudaDeviceSynchronize();
        if (err != cudaSuccess) Stop("error in kernel launch");

        err = cudaMemcpy(C,_C,size*sizeof(double),cudaMemcpyDeviceToHost);
        if (err != cudaSuccess) Stop("invalid copy -C-");
```



```
// ***************************************************************************
__global__ void vectorAddGPU(const int size, double* _A, double* _B, double* _C)
// ***************************************************************************
{
        const int i = blockDim.x*blockIdx.x + threadIdx.x;
        if (i < size)
        {
                _C[i] = _A[i] + _B[i];
        }
}
```



```
        // GPU deallocations :
        cudaFree(_A); _A = NULL;
        cudaFree(_B); _B = NULL;
        cudaFree(_C); _C = NULL;
```

Piece of advice

❑ Avoid threads running in parallel to write at the same memory position (memory conflict).

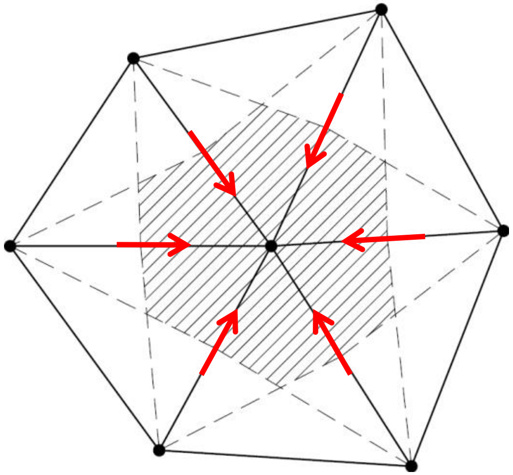
❑ Threads from the same warp should access to the same device memory segment, since access to a 128-byte device memory segment can be performed within a single memory transaction.

Use Shared, constant and/or texture memory when possible.

❑ Be careful with *if statements* – avoid thread divergence.

❑ If it is possible, use single precision instead of double precision arithmetic. In Fermi GPUs, single precision operation rate is 2x higher than the double precision one.

❑ Use all the available resources (GPU + CPU).



Piece of advice

□ Avoid threads running in parallel to write at the same memory position (memory conflict).

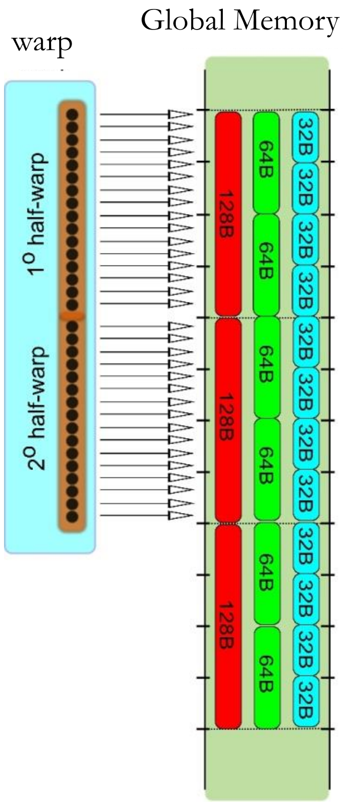
□ Threads from the same warp should access to the same device memory segment, since access to a 128-byte device memory segment can be performed within a single memory transaction.

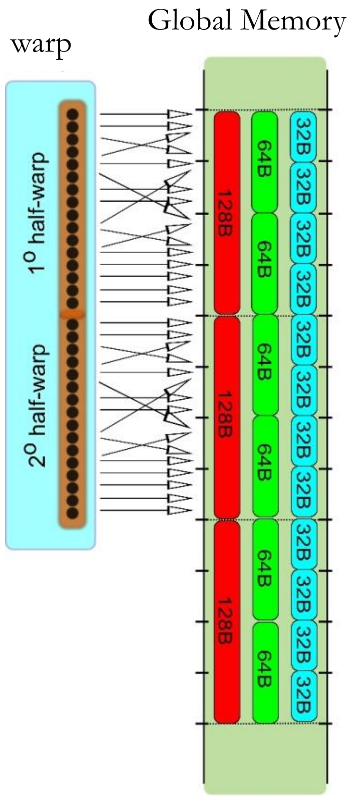
Use Shared, constant and/or texture memory when possible.

□ Be careful with *if statements* – avoid thread divergence.

□ If it is possible, use single precision instead of double precision arithmetic. In Fermi GPUs, single precision operation rate is 2x higher than the double precision one.

□ Use all the available resources (GPU + CPU).





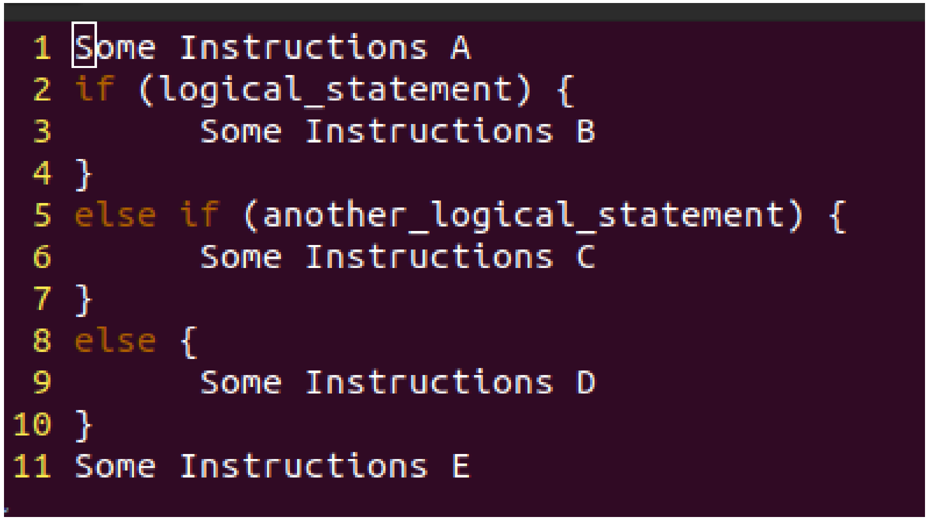
Piece of advice

❑ Avoid threads running in parallel to write to the same memory position (memory conflict).

❑ Threads from the same warp should access to the same device memory segment, since access to a 128-byte device memory segment can be performed within a single memory transaction.

Use Shared, constant and/or texture memory when possible.

❑ Be careful with *if statements* – avoid thread divergence.



```
Some Instructions A
if (logical_statement) {
        Some Instructions B
}
else if (another_logical_statement) {
        Some Instructions C
}
else {
        Some Instructions D
}
Some Instructions E
```

❑ If it is possible, use single precision instead of double precision arithmetic. In Fermi GPUs, single precision operation rate is 2x higher than the double precision one.

❑ Use all the available resources (GPU + CPU).

Piece of advice

□ Avoid threads running in parallel to write to the same memory position (memory conflict).

□ Threads from the same warp should access to the same device memory segment, since access to a 128-byte device memory segment can be performed within a single memory transaction.

Use Shared, constant and/or texture memory when possible.

□ Be careful with *if statements* – avoid thread divergence.

□ If it is possible, use single precision instead of double precision arithmetic. In Fermi GPUs, single precision operation rate is 2x higher than the double precision one.

□ Use all the available resources (GPU + CPU).

$$\overbrace{\underbrace{\frac{\partial \vec{R}}{\partial \vec{U}}}_{\textbf{SPA}} \Delta \vec{U} = \underbrace{-R\left(\vec{U}\right)}_{\textbf{DPA}}}^{\textbf{MPA}}$$

$$\vec{U}^{n+1} = \vec{U}^{n} + \Delta \vec{U}$$

Piece of advice

❑ Avoid threads running in parallel to write to the same memory position (memory conflict).

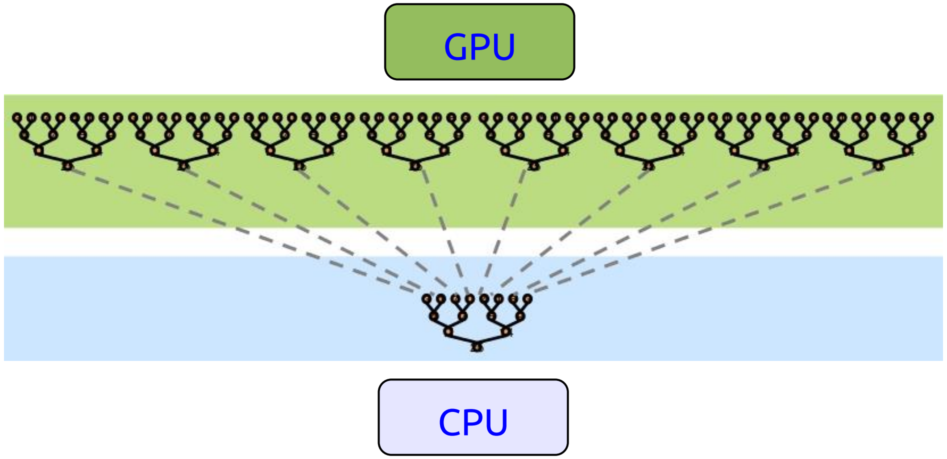
❑ Threads from the same warp should access to the same device memory segment, since access to a 128-byte device memory segment can be performed within a single memory transaction.

Use Shared, constant and/or texture memory when possible.

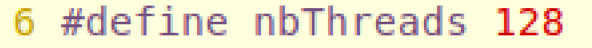
❑ Be careful with *if statements* – avoid thread divergence.

❑ If it is possible, use single precision instead of double precision arithmetic. In Fermi GPUs, single precision operation rate is 2x higher than the double precision one.

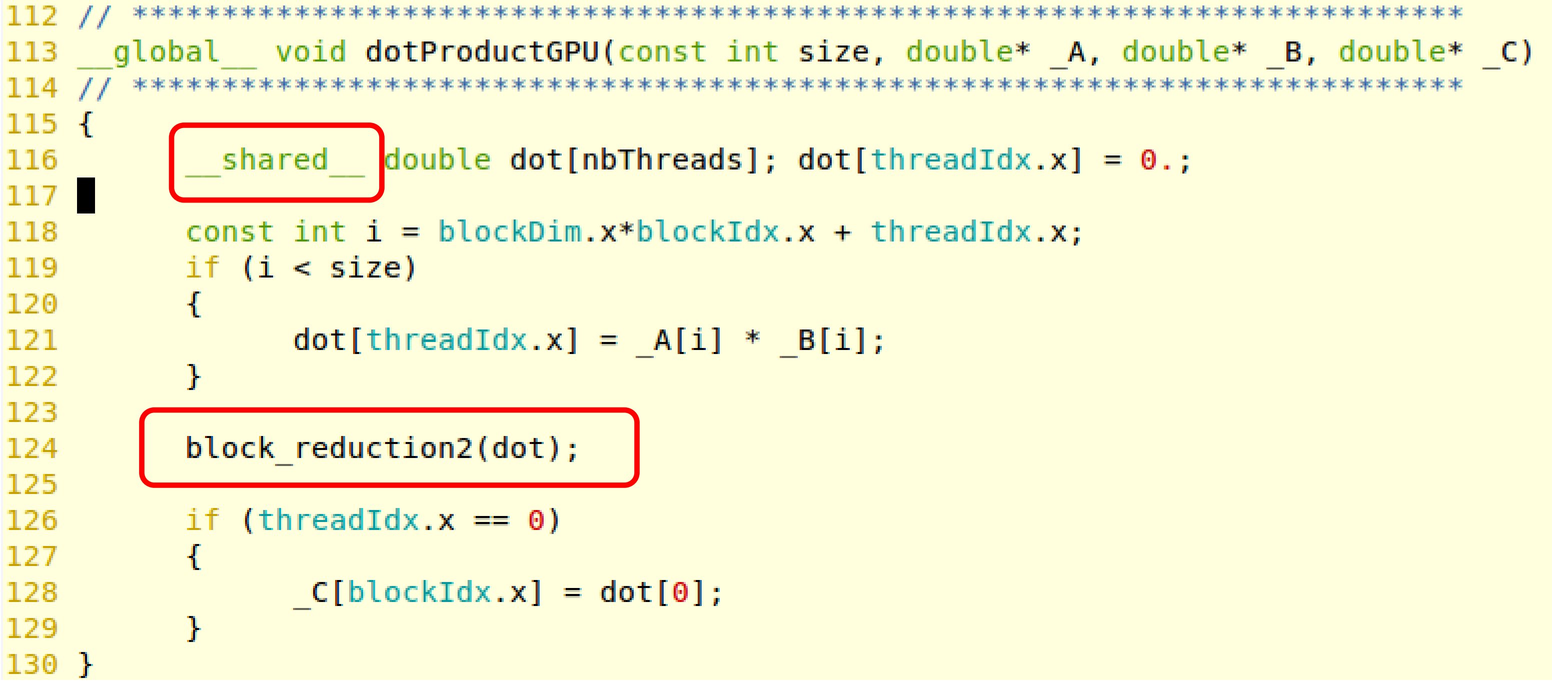
❑ Use all the available resources (GPU + CPU).



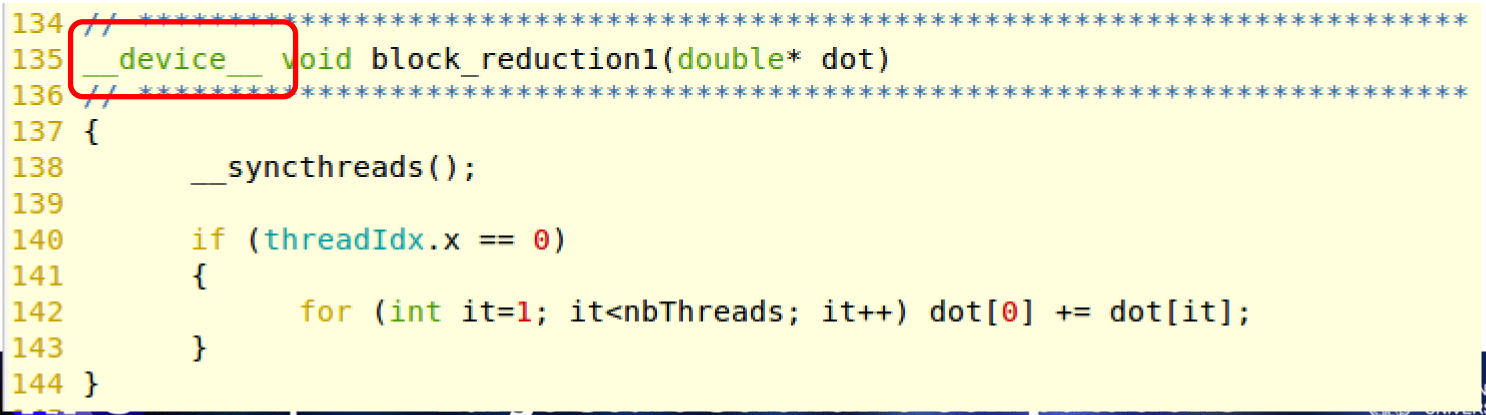
Dot product



```
#define nbThreads 128
```

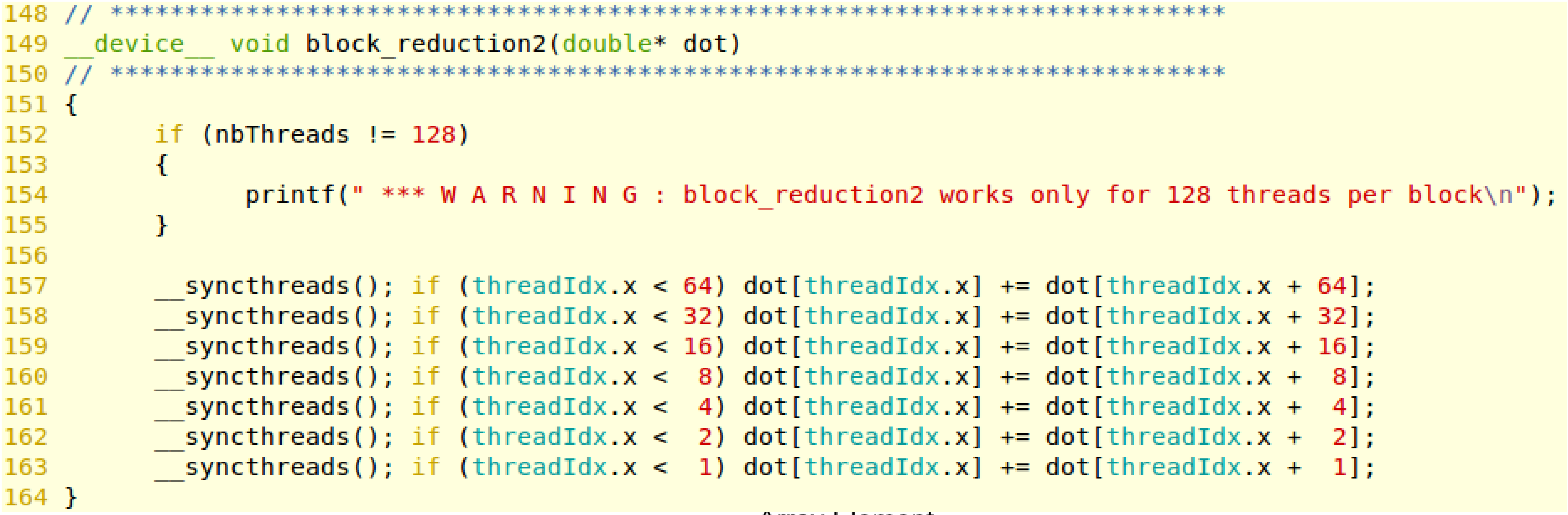


```
// ***************************************************************************
__global__ void dotProductGPU(const int size, double* _A, double* _B, double* _C)
// ***************************************************************************
{
	__shared__ double dot[nbThreads]; dot[threadIdx.x] = 0.;

	const int i = blockDim.x*blockIdx.x + threadIdx.x;
	if (i < size)
	{
		dot[threadIdx.x] = _A[i] * _B[i];
	}

	block_reduction2(dot);

	if (threadIdx.x == 0)
	{
		_C[blockIdx.x] = dot[0];
	}
}
```

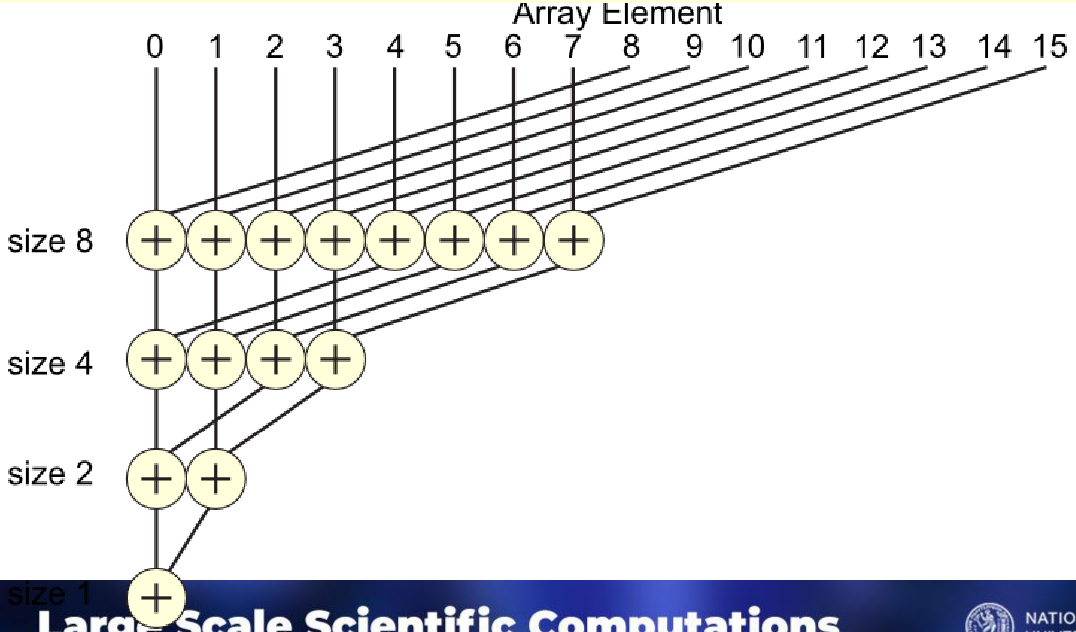


```
// ***************************************************************************
__device__ void block_reduction1(double* dot)
// ***************************************************************************
{
	__syncthreads();

	if (threadIdx.x == 0)
	{
		for (int it=1; it<nbThreads; it++) dot[0] += dot[it];
	}
}
```

Dot product



```
// ***************************************************************************
__device__ void block_reduction2(double* dot)
// ***************************************************************************
{
      if (nbThreads != 128)
      {
            printf(" *** W A R N I N G : block_reduction2 works only for 128 threads per block\n");
      }

      __syncthreads(); if (threadIdx.x < 64) dot[threadIdx.x] += dot[threadIdx.x + 64];
      __syncthreads(); if (threadIdx.x < 32) dot[threadIdx.x] += dot[threadIdx.x + 32];
      __syncthreads(); if (threadIdx.x < 16) dot[threadIdx.x] += dot[threadIdx.x + 16];
      __syncthreads(); if (threadIdx.x <  8) dot[threadIdx.x] += dot[threadIdx.x +  8];
      __syncthreads(); if (threadIdx.x <  4) dot[threadIdx.x] += dot[threadIdx.x +  4];
      __syncthreads(); if (threadIdx.x <  2) dot[threadIdx.x] += dot[threadIdx.x +  2];
      __syncthreads(); if (threadIdx.x <  1) dot[threadIdx.x] += dot[threadIdx.x +  1];
}
```



Summary

- ❑ __global__ : GPU function launched by the host (kernel)
- ❑ __device__ : GPU function launched by the device
- ❑ __host__ : CPU function launched by the host

- ❑ __shared__ : Variable in the shared memory

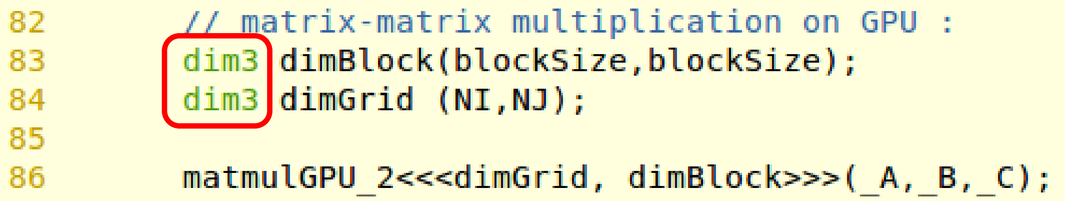
- ❑ __syncthreads() : Block thread synchronization
- ❑ cudaDeviceSynchronize() : CPU-GPU synchronization

- ❑ cudaError_t cudaMalloc(void** ptr, size_t size);

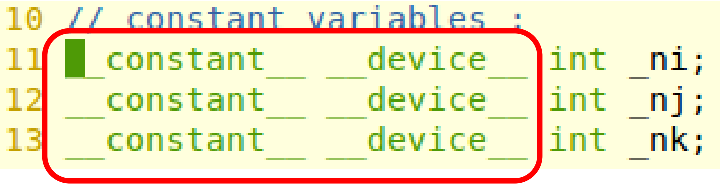
- ❑ cudaError_t cudaFree(void* ptr);

- ❑ cudaError_t cudaMemcpy(void* destination, void* source, size_t size, cudaMemcpyKind kind);

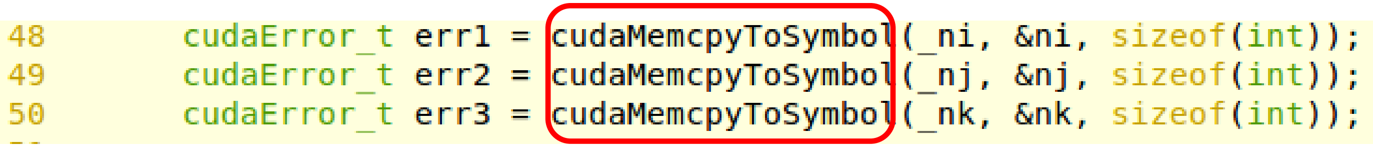
Matrix-matrix multiplication



```
        // matrix-matrix multiplication on GPU :
        dim3 dimBlock(blockSize,blockSize);
        dim3 dimGrid (NI,NJ);

        matmulGPU_2<<<dimGrid, dimBlock>>>(_A,_B,_C);
```

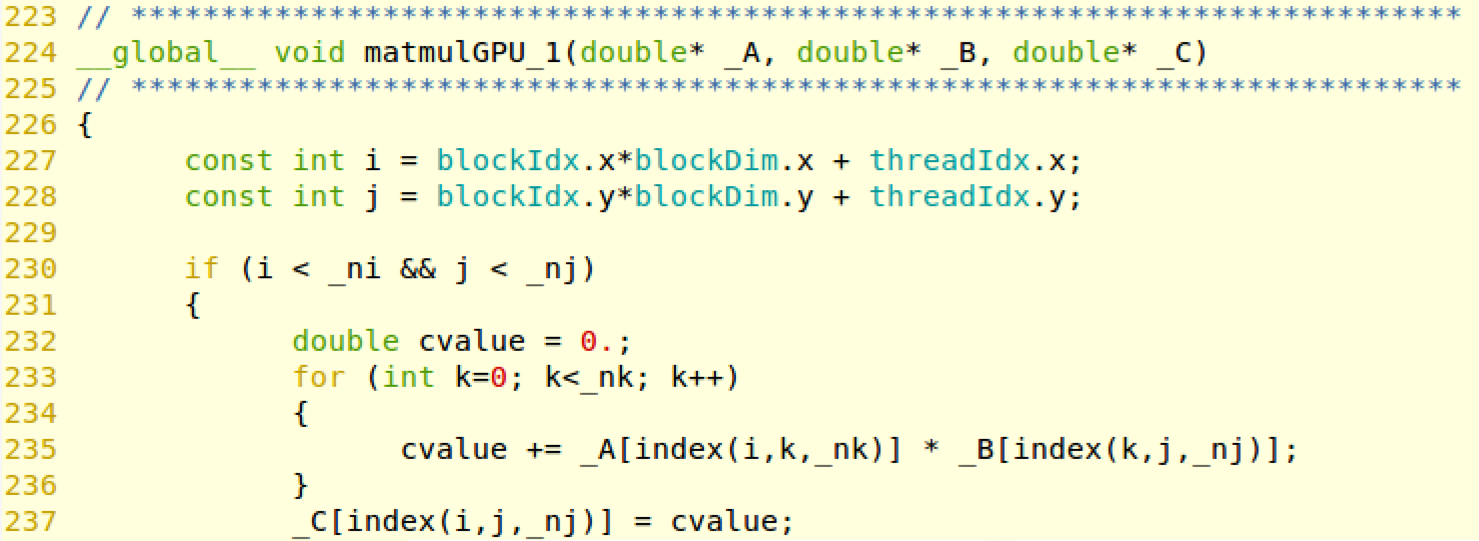


```
// constant variables :
__constant__ __device__ int _ni;
__constant__ __device__ int _nj;
__constant__ __device__ int _nk;
```

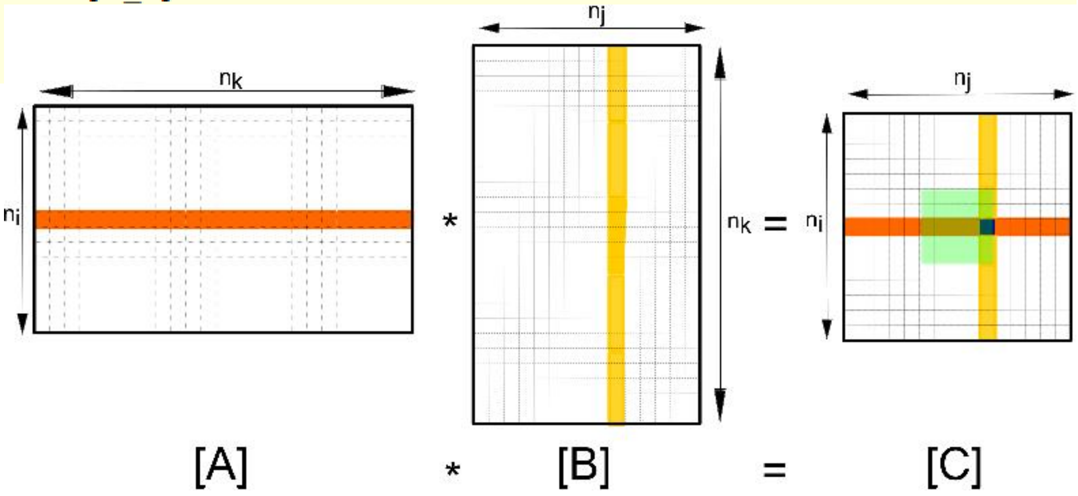


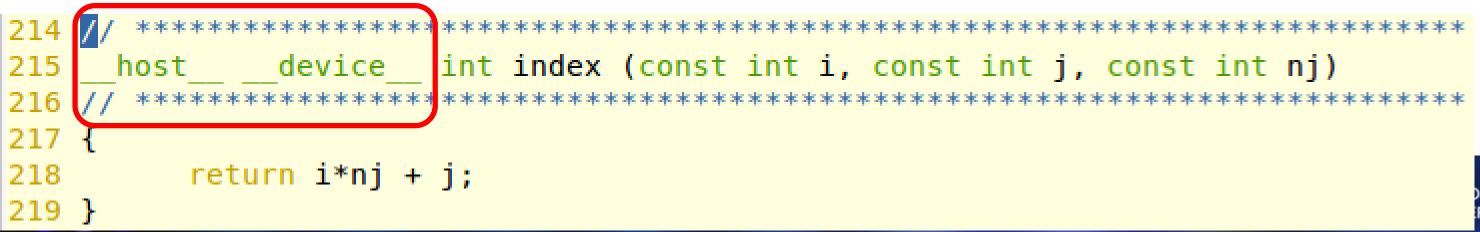
```
        cudaError_t err1 = cudaMemcpyToSymbol(_ni, &ni, sizeof(int));
        cudaError_t err2 = cudaMemcpyToSymbol(_nj, &nj, sizeof(int));
        cudaError_t err3 = cudaMemcpyToSymbol(_nk, &nk, sizeof(int));
```

Matrix-matrix multiplication



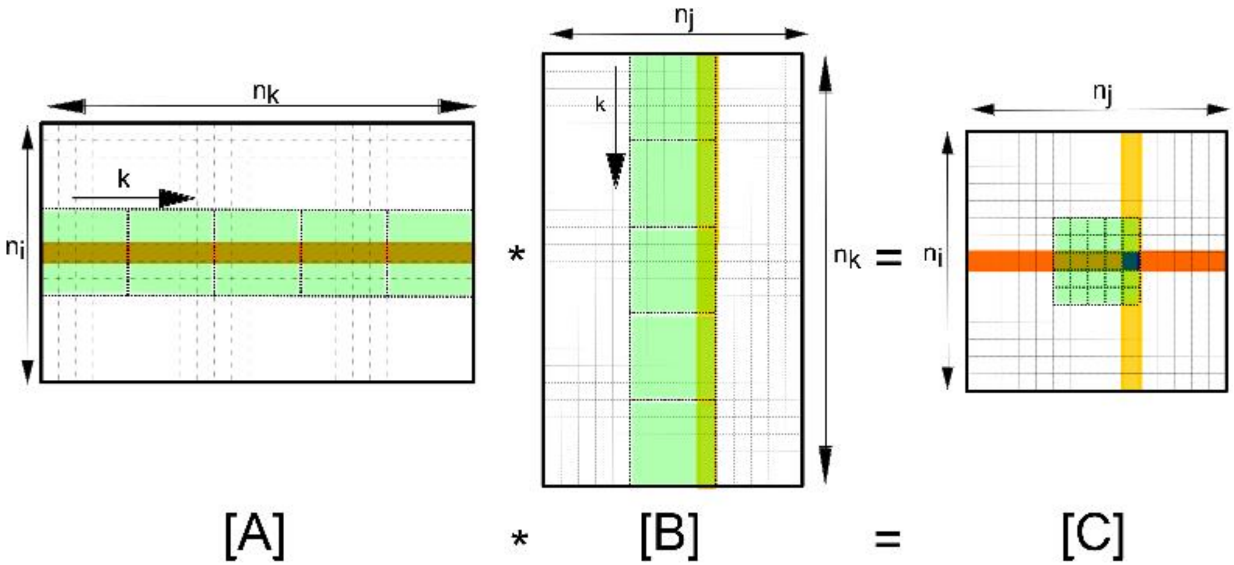
```
// ***************************************************************************
__global__ void matmulGPU_1(double* _A, double* _B, double* _C)
// ***************************************************************************
{
      const int i = blockIdx.x*blockDim.x + threadIdx.x;
      const int j = blockIdx.y*blockDim.y + threadIdx.y;

      if (i < _ni && j < _nj)
      {
            double cvalue = 0.;
            for (int k=0; k<_nk; k++)
            {
                  cvalue += _A[index(i,k,_nk)] * _B[index(k,j,_nj)];
            }
            _C[index(i,j,_nj)] = cvalue;
      }
}
```



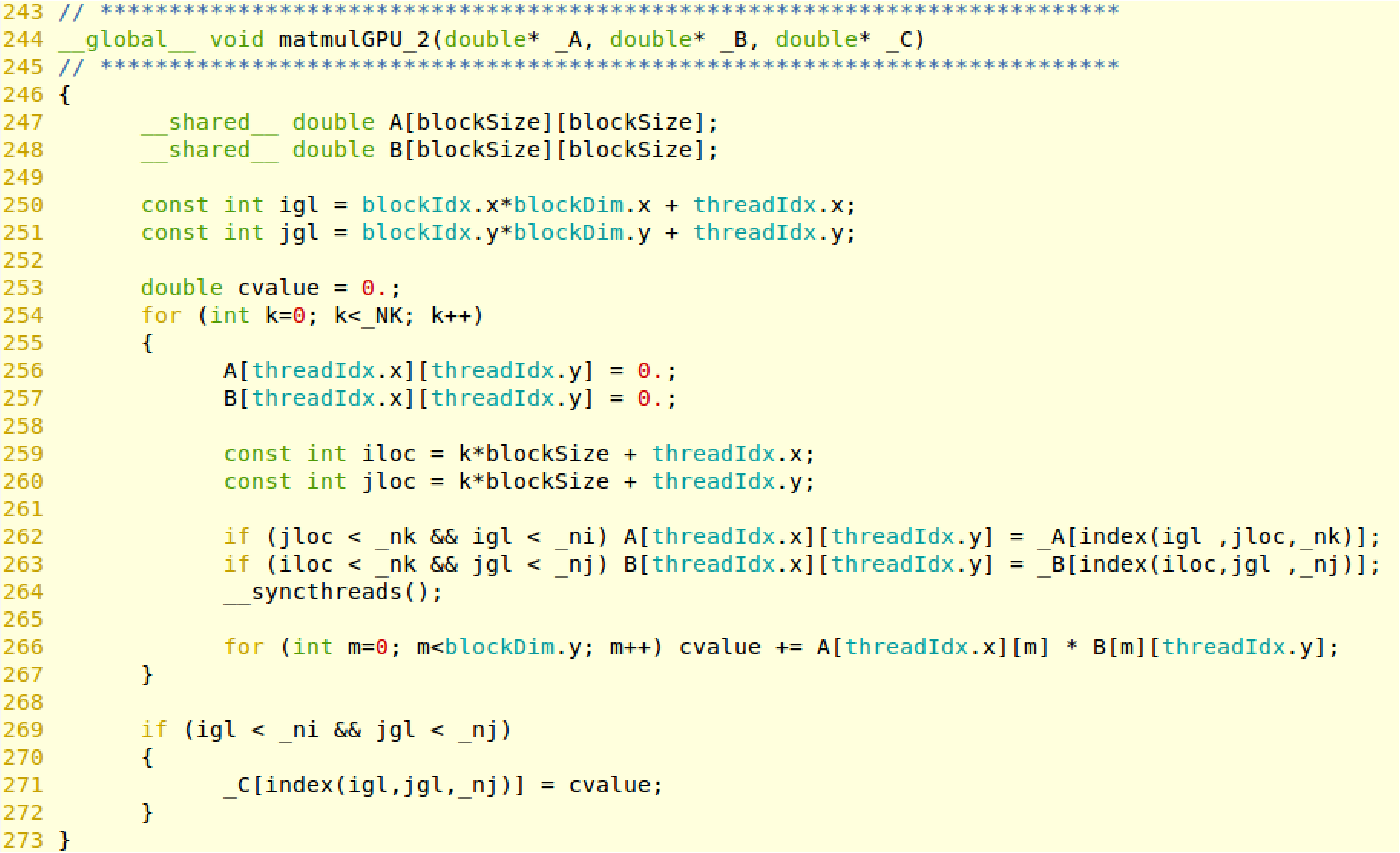


```
// ***************************************************************************
__host__ __device__ int index (const int i, const int j, const int nj)
// ***************************************************************************
{
      return i*nj + j;
}
```

Matrix-matrix multiplication



Matrix-matrix multiplication



```
// *************************************************************************
__global__ void matmulGPU_2(double* _A, double* _B, double* _C)
// *************************************************************************
{
      __shared__ double A[blockSize][blockSize];
      __shared__ double B[blockSize][blockSize];

      const int igl = blockIdx.x*blockDim.x + threadIdx.x;
      const int jgl = blockIdx.y*blockDim.y + threadIdx.y;

      double cvalue = 0.;
      for (int k=0; k<_NK; k++)
      {
            A[threadIdx.x][threadIdx.y] = 0.;
            B[threadIdx.x][threadIdx.y] = 0.;

            const int iloc = k*blockSize + threadIdx.x;
            const int jloc = k*blockSize + threadIdx.y;

            if (jloc < _nk && igl < _ni) A[threadIdx.x][threadIdx.y] = _A[index(igl ,jloc,_nk)];
            if (iloc < _nk && jgl < _nj) B[threadIdx.x][threadIdx.y] = _B[index(iloc,jgl ,_nj)];
            __syncthreads();

            for (int m=0; m<blockDim.y; m++) cvalue += A[threadIdx.x][m] * B[m][threadIdx.y];
      }

      if (igl < _ni && jgl < _nj)
      {
            _C[index(igl,jgl,_nj)] = cvalue;
      }
}
```