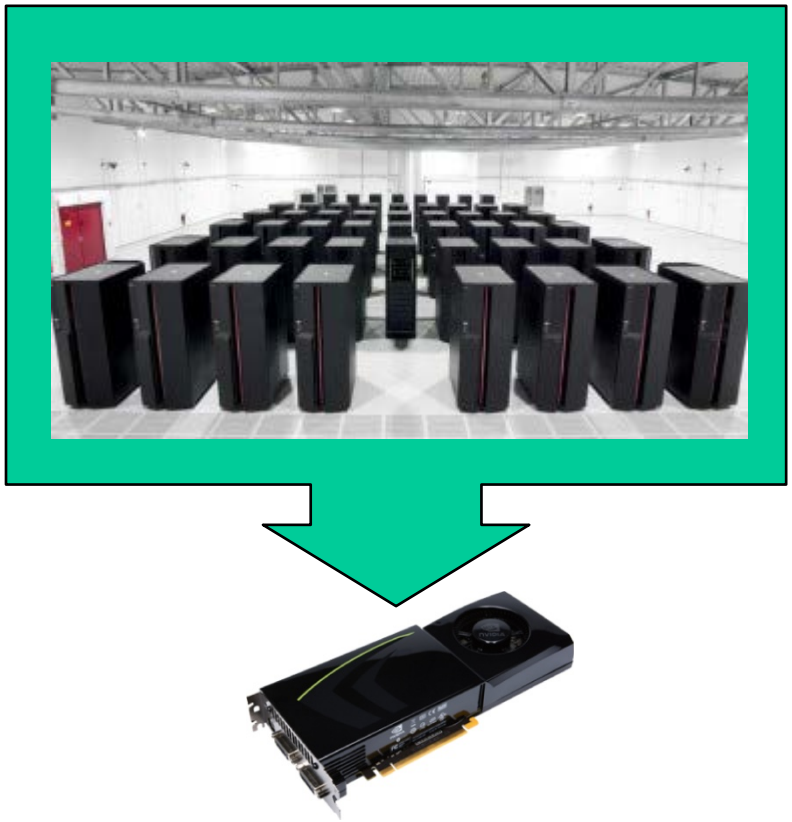
Introduction to modern graphic processing unit (GPU) architecture and programming in CUDA

Xenofon Trompoukis, Dr Mechanical engineer

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

Why GPUs ?

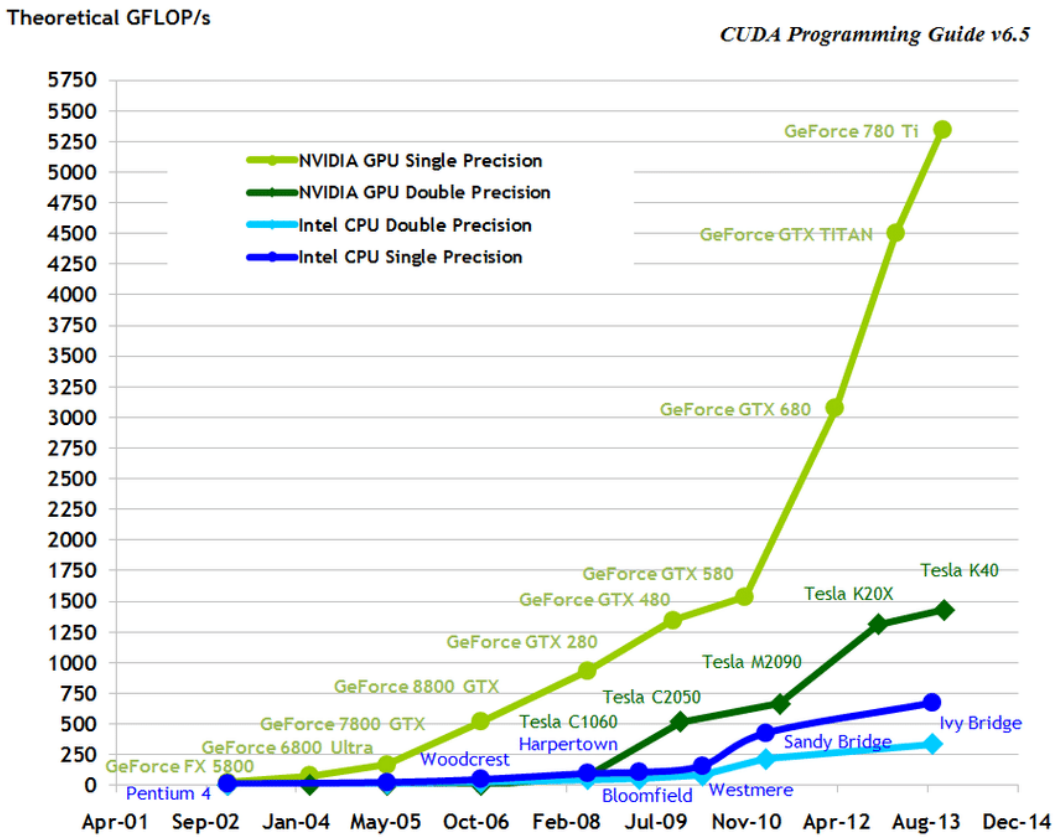
- Parallel processing units
- High floating-point operations rate (double and single precision arithmetic)
- GPU embedded, low latency, RAM
- Various programming environments
- Low cost & energy consumption compared to their computational power



EURO Greece

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 compared to their computational power



EURO Greece

Why GPUs ?

❑ Parallel processing units

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

NVIDIA A100 (Ampere, 2020)
- ✓ 19.5 TFLOPS (single precision)
- ✓ 9.7 TFLOPS (double precision)
- ✓ 156 / 19.5 TFLOPS with Tensor cores

❑ GPU embedded, low latency, RAM

❑ Various programming environments

❑ Low cost & energy consumption
compared to their computational power

EURO
Greece

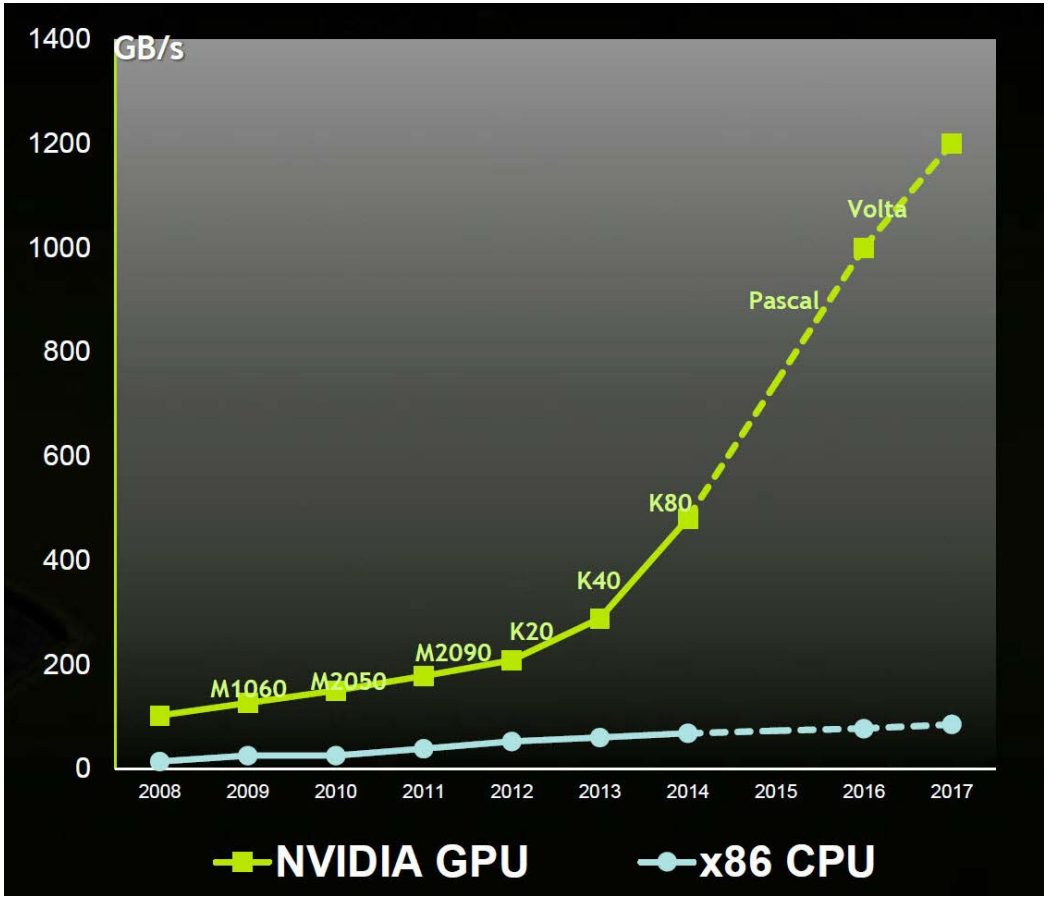
Why GPUs ?

- ❑ Parallel processing units
- ❑ High floating-point operations rate (double and single precision arithmetic)
- ❑ GPU embedded, low latency, RAM

NVIDIA A100 (Ampere, 2020)

- ✓ 40 GB of RAM
- ✓ Peak memory bandwidth: 1.6TB/s

- ❑ Various programming environments
- ❑ Low cost & energy consumption compared to their computational power



EURO Greece

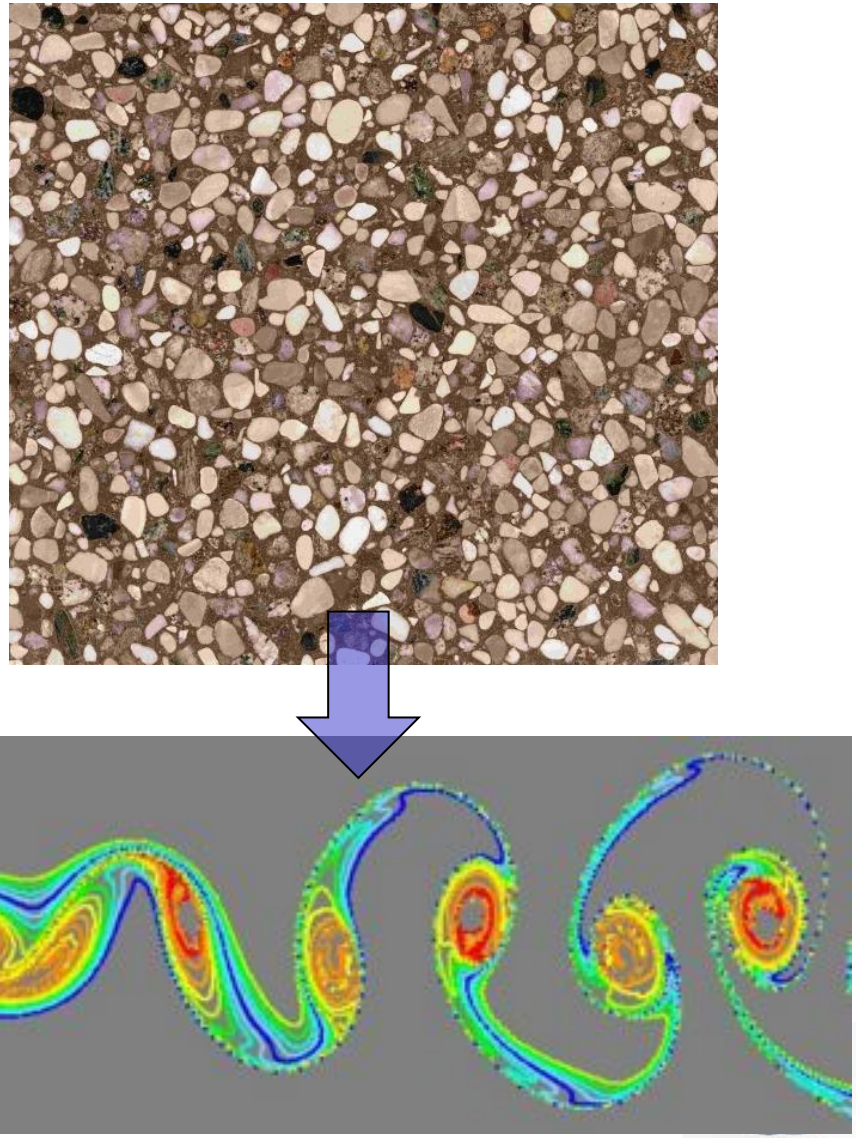
Why GPUs ?

- ❑ 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 compared to their computational power

OpenCL: Cross platform implementation
- C++
- Python

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

OpenACC



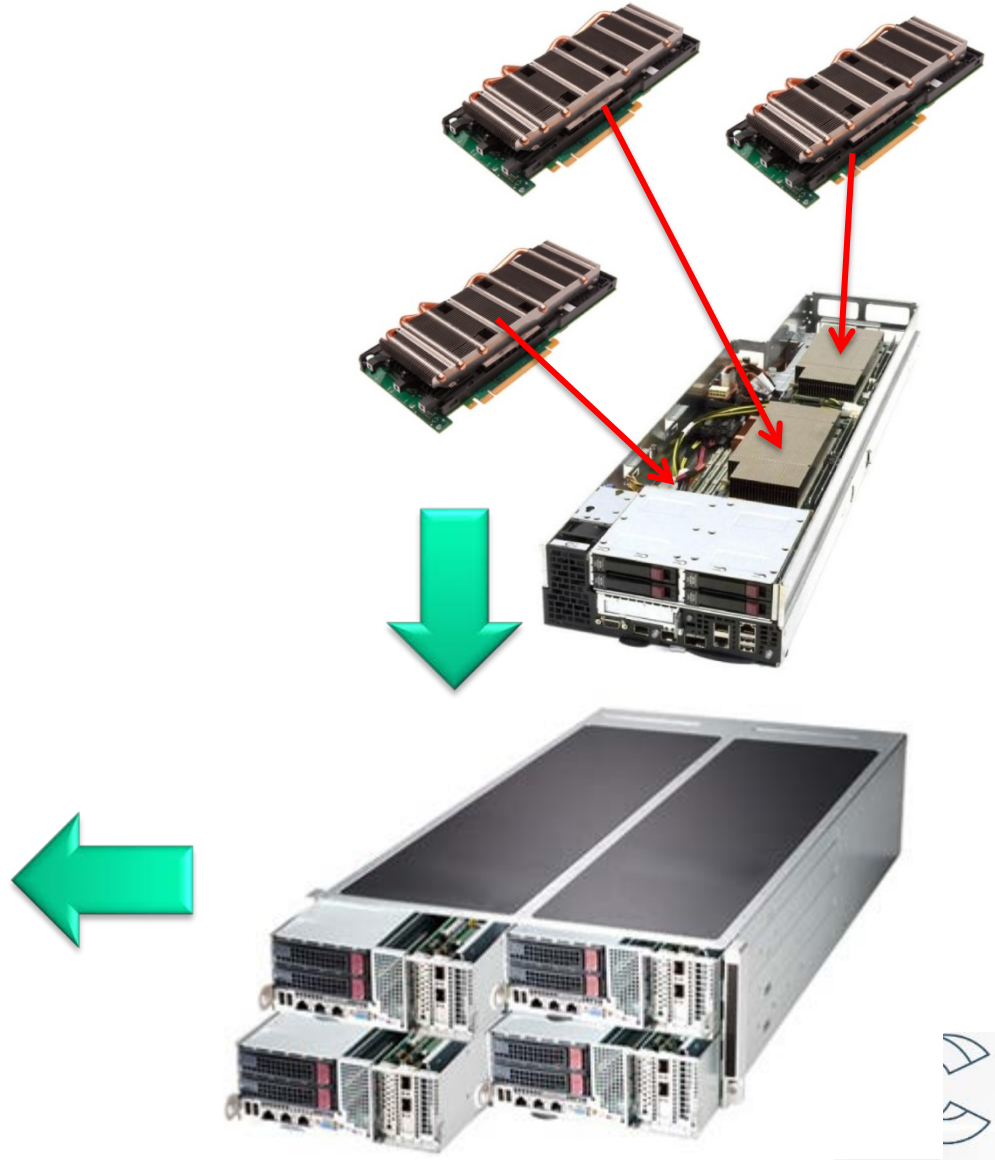
Why GPUs ?

- ❑ 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 compared to their computing power

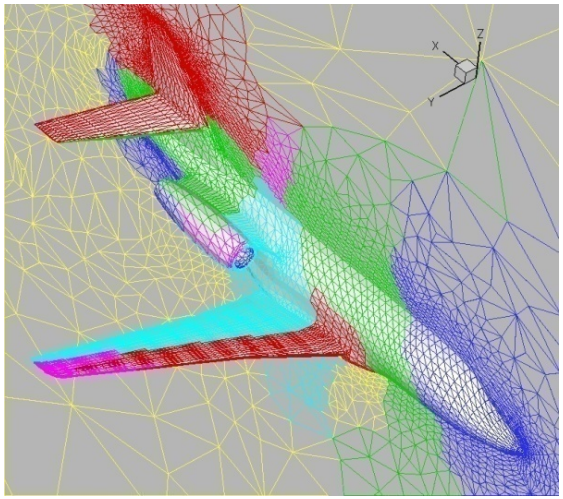
EURO
Greece

Why GPUs ?

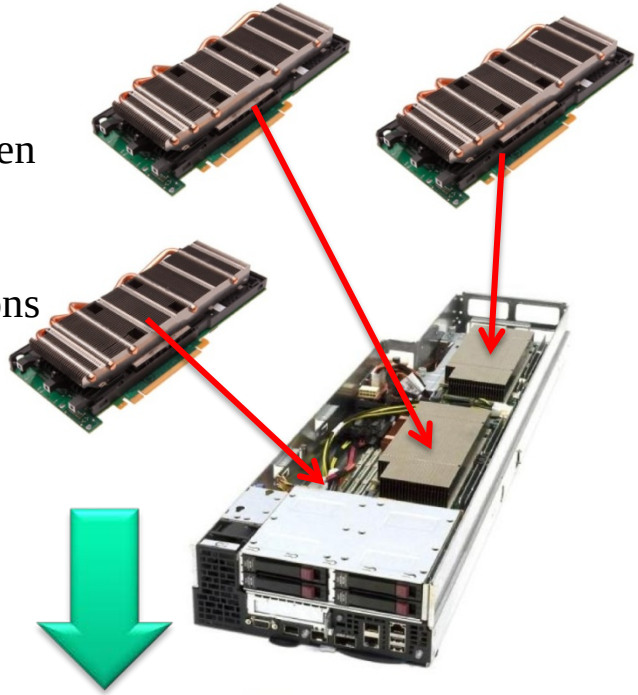
- Parallel processing units
- High floating-point operations rate (double and single precision arithmetic)
- GPU embedded, low latency, RAM
- Various programming environments
- Low cost & energy consumption based on their computational power



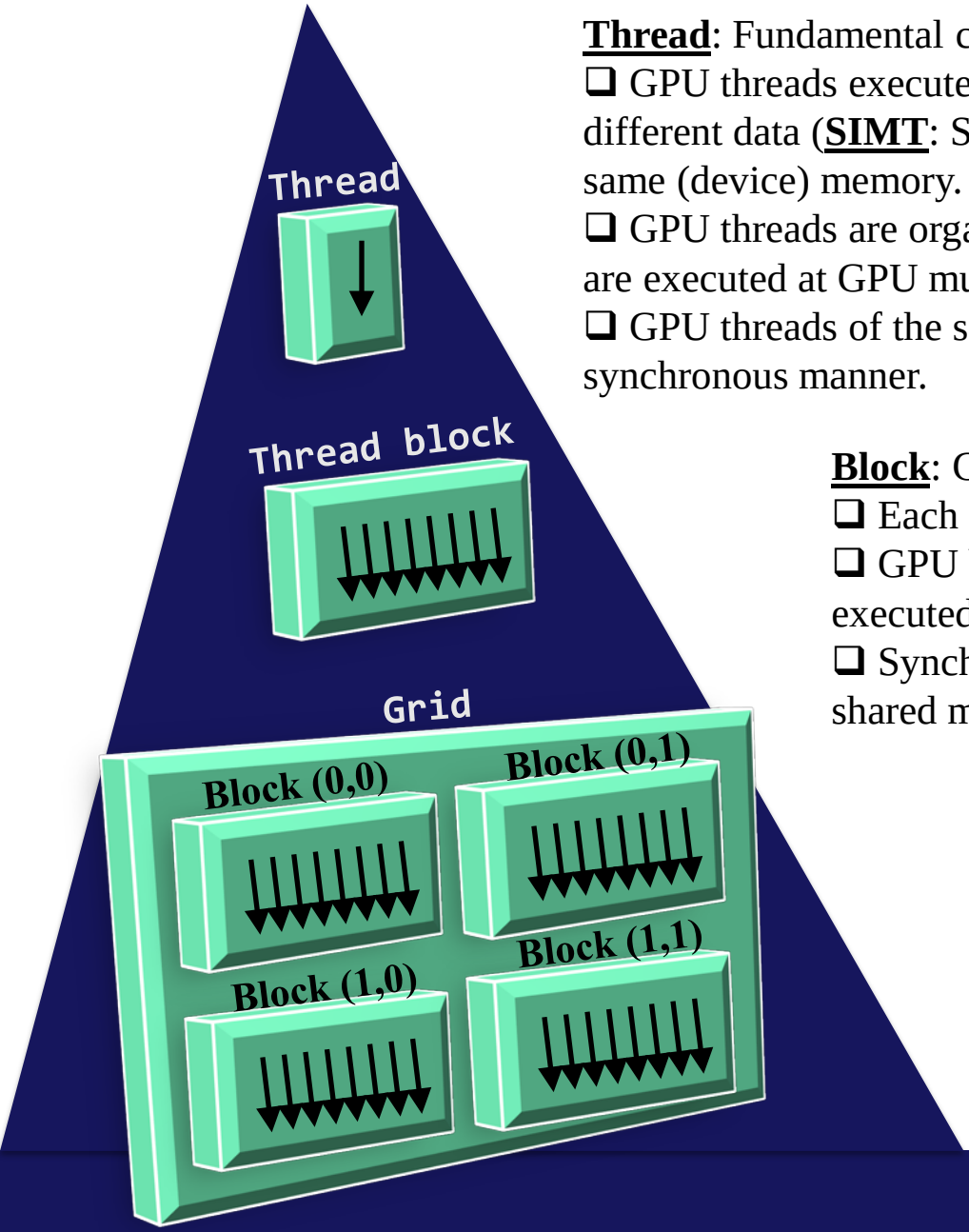
Why GPUs ?



- 1 sub-domain per GPU.
- MPI → data transactions between GPUs on different compute nodes.
- Host memory → data transactions between GPUs on the same node.



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 organized into **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

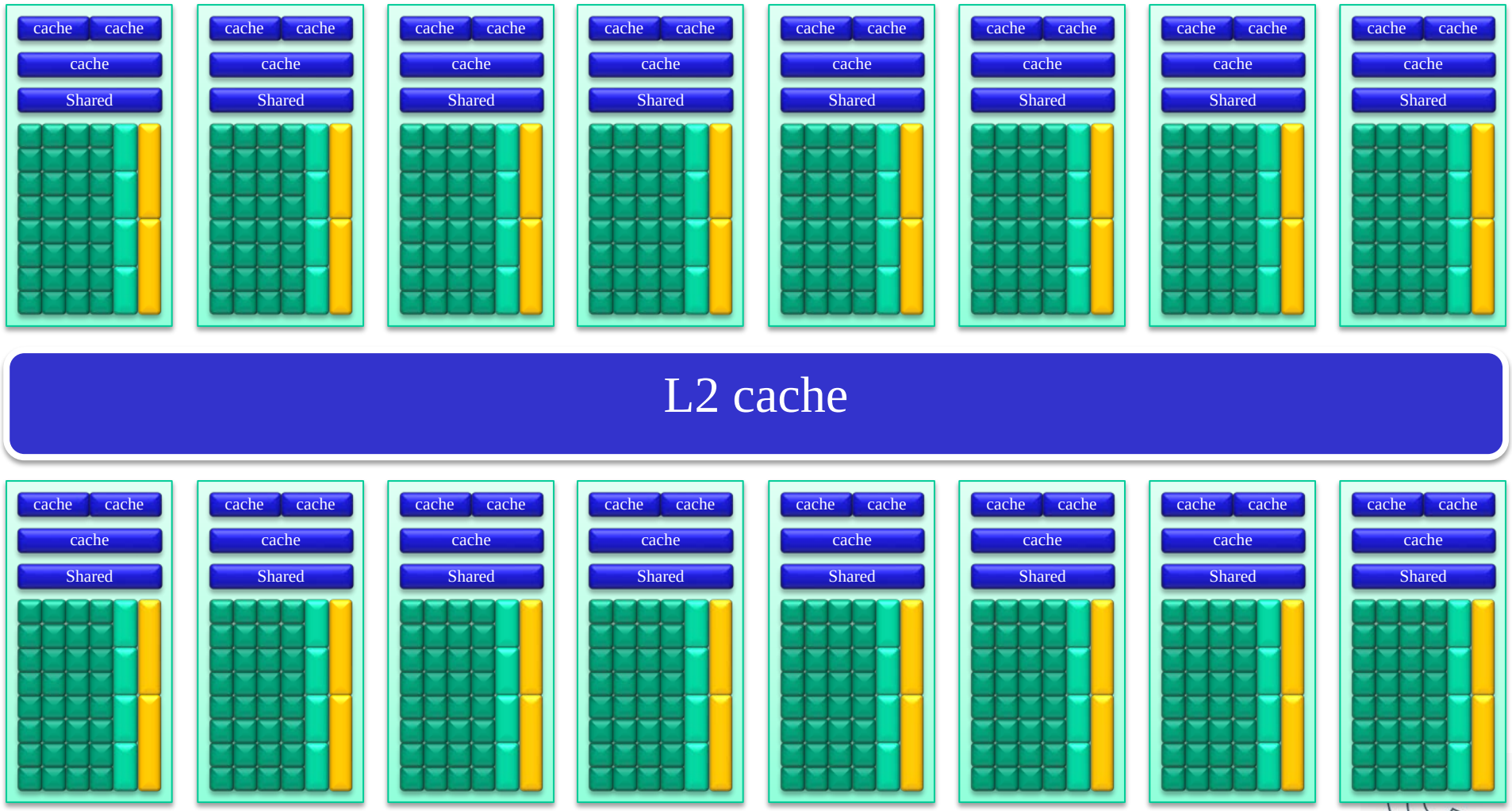
Grid: Cluster of thread blocks

✓ A100 can execute up to 221,184 threads in parallel

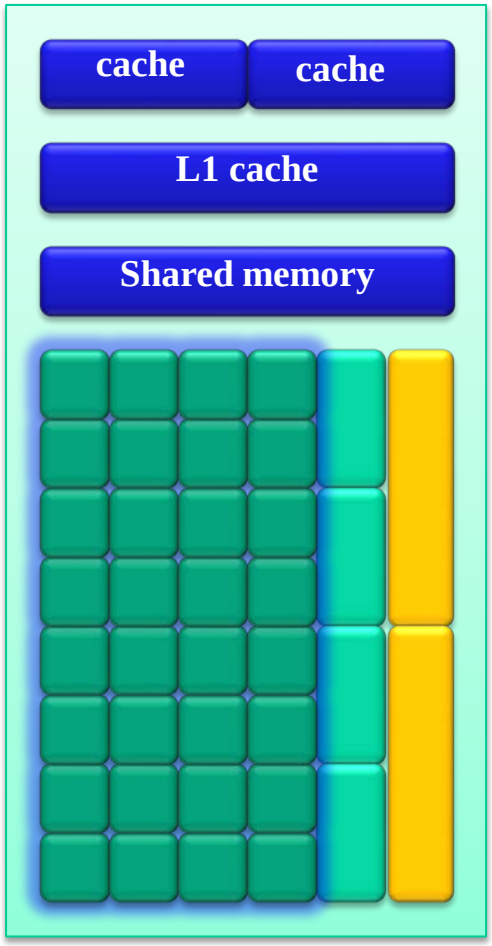
EURO Greece

GPU Architecture

Fermi (2010):16 streaming multiprocessors (SMs)



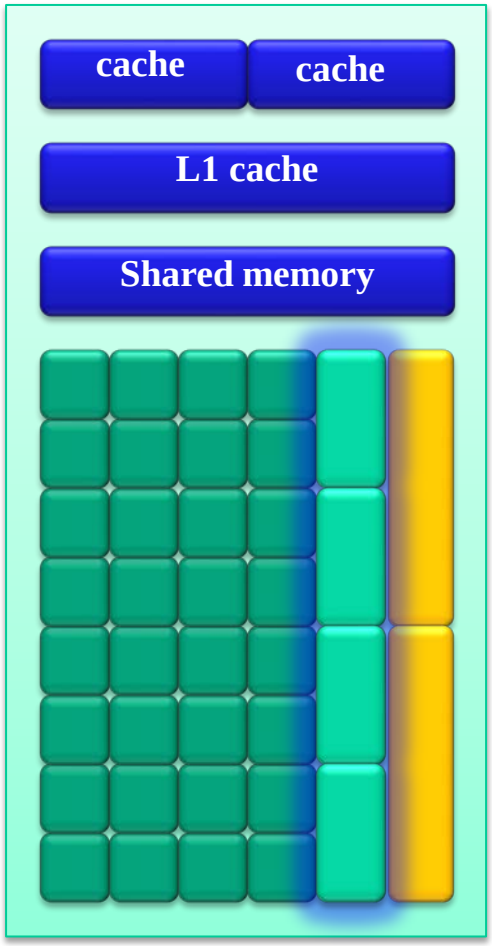
GPU Architecture



Each Fermi SM consists of:

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

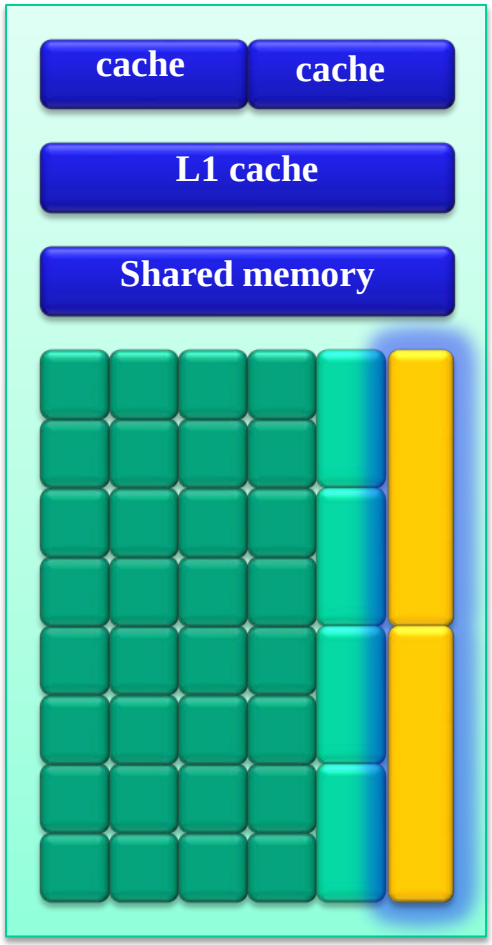
GPU Architecture



Each Fermi SM consists of:

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

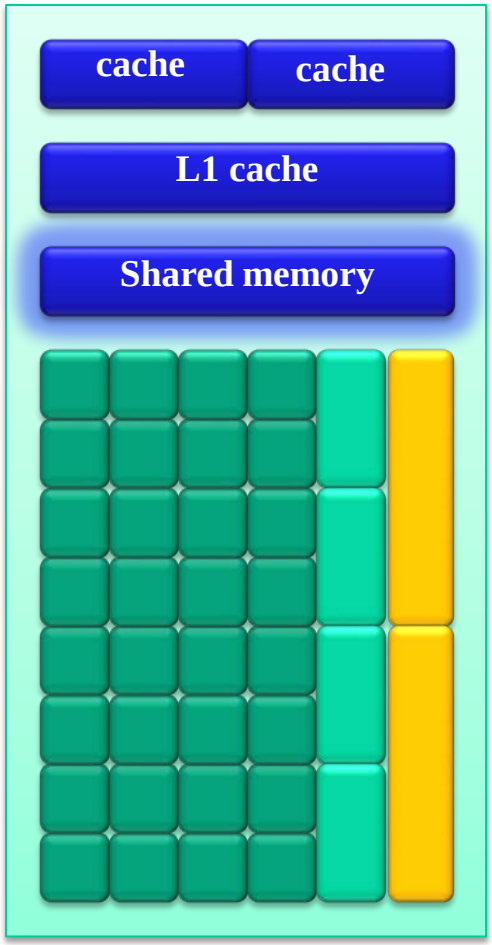
GPU Architecture



Each Fermi SM consists of:

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

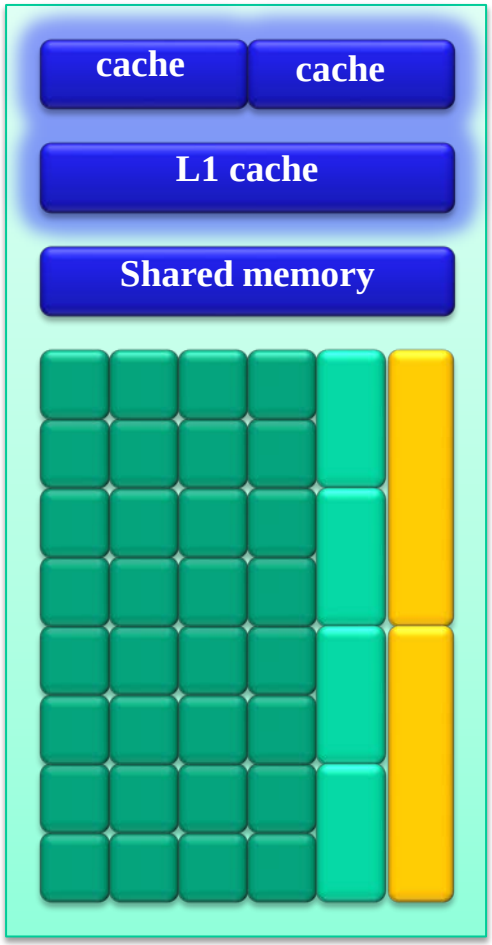
GPU Architecture



Each Fermi SM consists of:

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

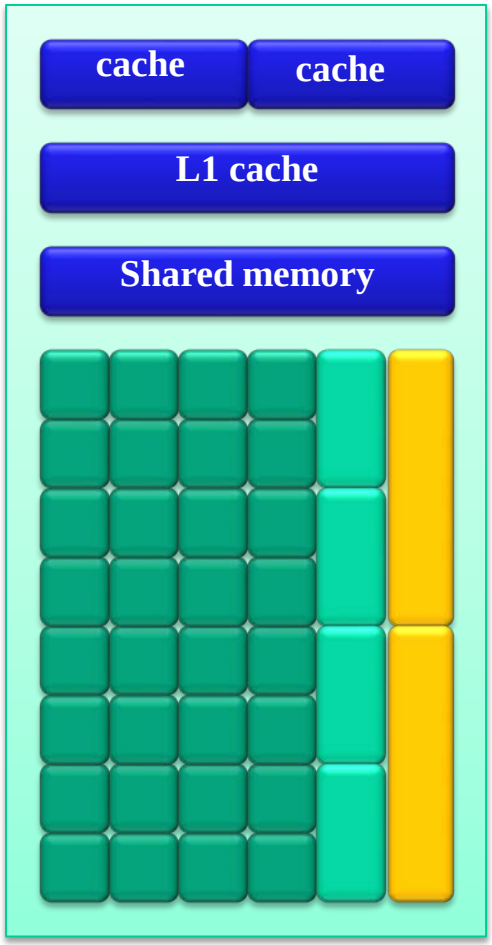
GPU Architecture



Each Fermi SM 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



	Fermi (2010)	Ampere (2020)
CUDA cores	32	128
Tensor cores	-	8
Shared memory & L1 cache	64 KB	192 KB
Texture cache	12 KB	12 KB
Constant cache	8 KB	8 KB
Registers	32,768 32-bit	65,536 32-bit
SMs	16	108 (A100)
Max. threads per SM	1,536	2,048
Max threads	24,576	221,184

Hello World: CPU implementation

```
#include <iostream>

int main()
{
      printf("# Hello world from the host\n");
      return 0;
}
```

Hello World: GPU implementation

```
#include <iostream>

__global__ void helloGPU(); // kernel declaration

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

      return 0;
}

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

Number of blocks into the grid

Number of threads into the block

Return type: void

**** This code compiles but it prints nothing in the screen

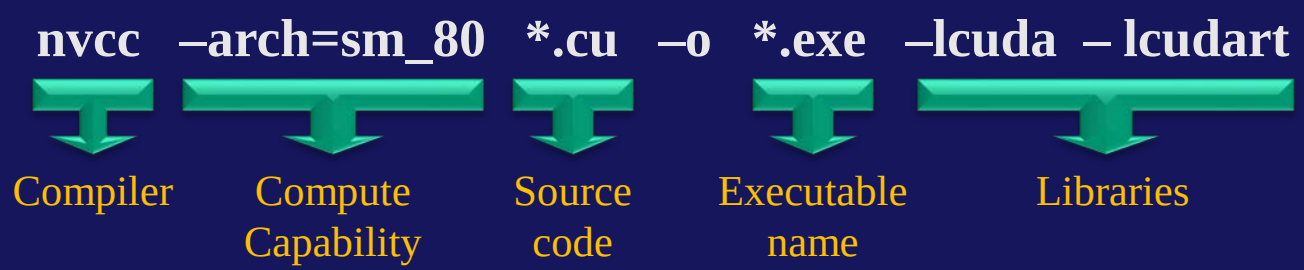
EURO Greece

Hello World: GPU implementation

```
#include <iostream>

__global__ void helloGPU(); // kernel declaration

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

      // synchronize host & device:
      cudaDeviceSynchronize();

      return 0;
}

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

Host-device synchronization

EURO Greece


nvcc –arch=sm_80 *.cu –o *.exe –lcuda – lcudart
Compiler
Compute Capability
Source code
Executable name
Libraries

Vector addition: CPU implementation

```cpp
#include <iostream>
#include <cmath>

int main()
{
	constexpr long size = 10;

	double* A = new double[size];
	double* B = new double[size];
	double* C = new double[size];

	for (auto i=0; i < size; i++) A[i] = (double)i;
	for (auto i=0; i < size; i++) B[i] = (double)i;
	for (auto i=0; i < size; i++) C[i] = NAN;

	for (auto i=0; i < size; i++) {
		C[i] = A[i] + B[i];
	}

	for (auto i=0; i < size; i++) {
		std::cout << A[i] << " + " << B[i] << " = " << C[i] << std::endl;
	}

	if (A != nullptr) delete[] A; A = nullptr;
	if (B != nullptr) delete[] B; B = nullptr;
	if (C != nullptr) delete[] C; C = nullptr;

	return 0;
}
```

EURO
Greece

Vector addition: GPU implementation (1/6)

```cpp
#include <cuda.h> // include CUDA header
#include <iostream>
#include <cmath>

int main()
{
	// STEP1: Allocate device memory and copy data to GPU
	// STEP2: Launch kernel(s)
	// STEP3: Copy results from device to host memory

	return 0;
}
```

EURO
Greece

Vector addition: GPU implementation (2/6)

```
// STEP1: Device memory allocations & initialization :
double* _A = (double*)device_alloc( (void*)A, size*sizeof(double), "A" );
double* _B = (double*)device_alloc( (void*)B, size*sizeof(double), "B" );
double* _C = (double*)device_alloc(           size*sizeof(double), "C" );

void* device_alloc(const long size, const std::string& varname)
{
      void* devptr = nullptr;
      cudaError_t err = cudaMalloc(&devptr, size /*SIZE IN BYTES*/);
      if (err != cudaSuccess) std::cerr << "Invalid allocation of \""+varname+"\"" << std::endl;
      if (err != cudaSuccess) exit(1);

      return devptr;
}

void* device_alloc(void* hostptr, const long size, const std::string& varname)
{
      void* devptr = device_alloc(size,varname);

      cudaError_t err = cudaMemcpy(devptr, hostptr, size, cudaMemcpyHostToDevice);
      if (err != cudaSuccess) std::cerr << "Invalid copy to \""+varname+"\"" << std::endl;
      if (err != cudaSuccess) exit(1);

      return devptr;
}
```

Allocate space in device memory

Memory copy

EURO Greece

Vector addition: GPU implementation (2/6)

```
// STEP1: Device memory allocations & initialization :
constexpr long size = 10;

double* A = new double[size];
double* B = new double[size];
double* C = new double[size];

for (auto i=0; i < size; i++) A[i] = (double)i;
for (auto i=0; i < size; i++) B[i] = (double)i;
for (auto i=0; i < size; i++) C[i] = NAN;

double* _A = (double*)device_alloc( (void*)A, size*sizeof(double), "A" );
double* _B = (double*)device_alloc( (void*)B, size*sizeof(double), "B" );
double* _C = (double*)device_alloc(           size*sizeof(double), "C" );
```

EURO Greece

Vector addition: GPU implementation (3/6)

```
// STEP3: Device memory deallocations :
if (_A != nullptr) cudaFree(_A); _A = nullptr;
if (_B != nullptr) cudaFree(_B); _B = nullptr;
if (_C != nullptr) cudaFree(_C); _C = nullptr;
```


```
if (A != nullptr) delete[] A; A = nullptr;
if (B != nullptr) delete[] B; B = nullptr;
if (C != nullptr) delete[] C; C = nullptr;
```

EURO
Greece

Vector addition: GPU implementation (4/6)

```
// STEP2: Launch kernel:
const int blockSize = 1024;
const int gridSize  = (size+blockSize-1)/blockSize;

vectorAddGPU <<< gridSize, blockSize >>> (size,_A,_B,_C);

cudaDeviceSynchronize();
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
      std::cerr << " Launch failure: " << cudaGetErrorString(err) << std::endl;
      exit(1);
}

// Copy results to host :
err = cudaMemcpy(C, _C, size*sizeof(double), cudaMemcpyDeviceToHost);
if (err != cudaSuccess) std::cerr << "Invalid device to host copy" << std::endl;
if (err != cudaSuccess) exit(1);

for (int i=0; i < size; i++) {
      std::cout << A[i] << " + " << B[i] << " = " << C[i] << std::endl;
}


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

Create as many threads as the size of the outputs

Is host-device synchronization needed here ?

Each thread processes a single element of arrays A, B

EURO Greece

Vector addition: GPU implementation (5/6)

```
int main()
{
      constexpr long size = 10;

      double* A = (double*)alloc_managed_mem(size*sizeof(double), "A");
      double* B = (double*)alloc_managed_mem(size*sizeof(double), "B");
      double* C = (double*)alloc_managed_mem(size*sizeof(double), "C");

      for (auto i=0; i < size; i++) A[i] = (double)i;
      for (auto i=0; i < size; i++) B[i] = (double)i;
      for (auto i=0; i < size; i++) C[i] = NAN;

      const int blockSize = 1024;
      const int gridSize  = (size+blockSize-1)/blockSize;
      vectorAddGPU <<< gridSize, blockSize >>> (size,A,B,C);

      cudaDeviceSynchronize();
      cudaError_t err = cudaGetLastError();
      if (err != cudaSuccess) {
            std::cerr << " Launch failure: " << cudaGetErrorString(err) << std::endl;
            exit(1);
      }

      for (auto i=0; i < size; i++) {
            std::cout << A[i] << " + " << B[i] << " = " << C[i] << std::endl;
      }

      if (A != nullptr) cudaFree(A); A = nullptr;
      if (B != nullptr) cudaFree(B); B = nullptr;
      if (C != nullptr) cudaFree(C); C = nullptr;

      return 0;
}
```

Maintaining pointers for both host and device memory makes GPU code too large. Wouldn't it be great to manage both using a single pointer?

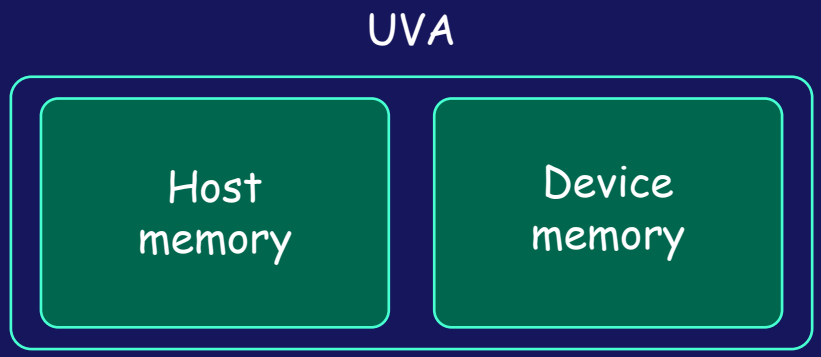
Is host-device synchronization needed here ?

Use cudaFree to free memory

EURO Greece

Vector addition: GPU implementation (6/6)

```
void* alloc_managed_mem(const long size, const std::string& varname)
{
      void* devptr = nullptr;
      cudaError_t err = cudaMallocManaged(&devptr, size /*SIZE IN BYTES*/);
      if (err != cudaSuccess) std::cerr << "Invalid allocation of \""+varname+"\"" << std::endl;
      if (err != cudaSuccess) exit(1);

      return devptr;
}

// CHECK whether the compute device supports managed memory:
int attr;
cudaDeviceGetAttribute(&attr, cudaDevAttrManagedMemory, 0);
if (attr == 0) std::cerr << "This device does not support managed memory" << std::endl;
if (attr == 0) exit(1);
```



EURO Greece

Piece of advice

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

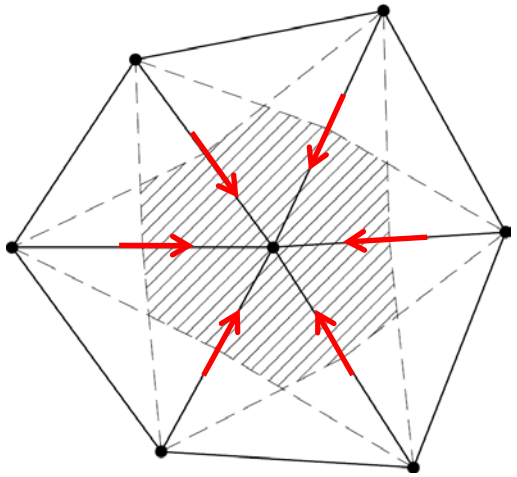
- Threads from the same warp should access to the same device memory segment, since access to a 128-byte device memory segment can be performed within a single memory transaction.

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).



EURO
Greece

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.

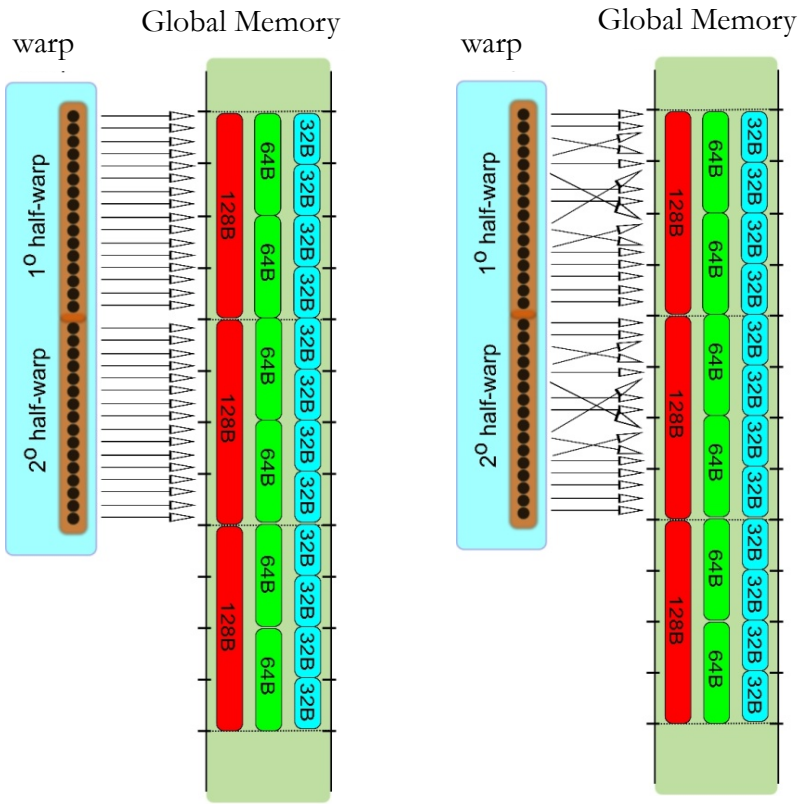
Use Shared, constant and/or texture memory when possible.

❑ 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).

Coalesced memory access



EURO
Greece

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).

```
1 Some Instructions A
2 if (logical_statement) {
3       Some Instructions B
4 }
5 else if (another_logical_statement) {
6       Some Instructions C
7 }
8 else {
9       Some Instructions D
10 }
11 Some Instructions E
```

EURO
Greece

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).

MPA

$$\frac{\partial \vec{R}}{\partial \vec{U}} \Delta \vec{U} = -R\left(\vec{U}\right)$$

SPA **DPA**

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

EURO Greece

Piece of advice

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

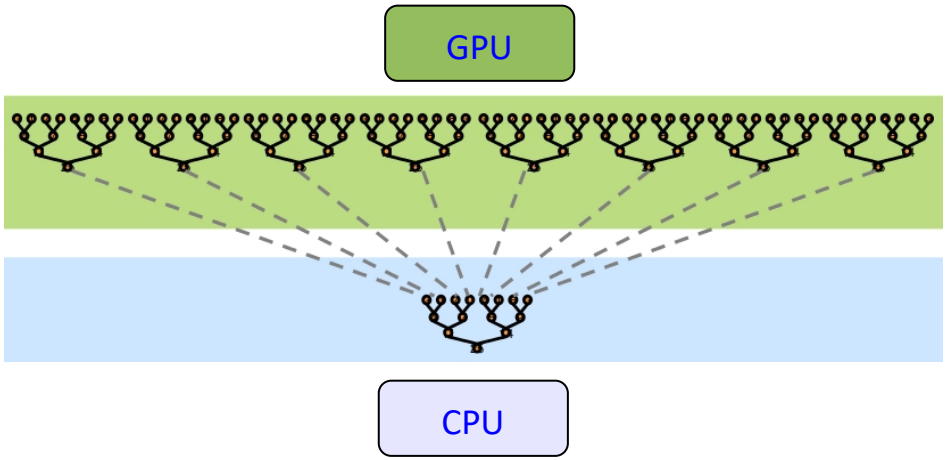
- Threads from the same warp should access to the same device memory segment, since access to a 128-byte device memory segment can be performed within a single memory transaction.

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).



EURO
Greece

Dot product: CPU implementation

```cpp
#include <iostream>

int main()
{
	constexpr long size = 10000;

	// Memory allocations & initialization :
	double* A = new double[size];
	double* B = new double[size];

	for (auto i=0; i < size; i++) A[i] = (double)i;
	for (auto i=0; i < size; i++) B[i] = (double)(size-i);

	// Dot product :
	double C = 0.0;
	for (auto i=0; i < size; i++) {
		C += A[i]*B[i];
	}
	std::cout << "DOT PRODUCT = " << C << std::endl;

	// Memory deallocations :
	if (A != nullptr) delete[] A; A = nullptr;
	if (B != nullptr) delete[] B; B = nullptr;

	return 0;
}
```

EURO
Greece

Dot product: GPU implementation (1/7)

```
__device__ double _C;
```

Static variable

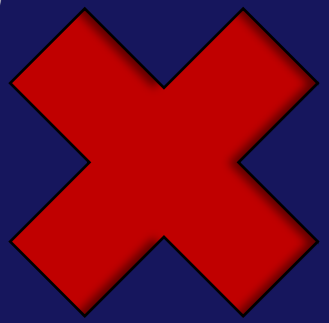
```
int main()
{
      constexpr long size = 10000;

      // Memory allocations & initialization :
      double* A = (double*)alloc_managed_mem(size*sizeof(double), "A");
      double* B = (double*)alloc_managed_mem(size*sizeof(double), "B");

      for (auto i=0; i < size; i++) A[i] = (double)i;
      for (auto i=0; i < size; i++) B[i] = (double)(size-i);

      // Compute dot product:
      double C = 0.0;
      cudaMemcpyToSymbol(_C,&C,sizeof(double));            // STEP1: Initializes _C

      constexpr int blockSize = 1024;
      constexpr int gridSize  = 1000;
      dotProductGPU <<< gridSize, blockSize >>> (size,A,B); // STEP2: Computes dot product, stored in _C

      cudaMemcpyFromSymbol(&C,_C,sizeof(double));          // STEP3: Copy result from device

      cudaError_t err = cudaGetLastError();
      if (err != cudaSuccess) std::cerr << " Launch failure: " << cudaGetErrorString(err) << std::endl;
      if (err != cudaSuccess) exit(1);

      std::cout << "DOT PRODUCT = " << C << std::endl;

      // Memory deallocations :
      if (A != nullptr) cudaFree(A); A = nullptr;
      if (B != nullptr) cudaFree(B); B = nullptr;

      return 0;
}
```

EURO Greece

Dot product: GPU implementation (2/7)

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

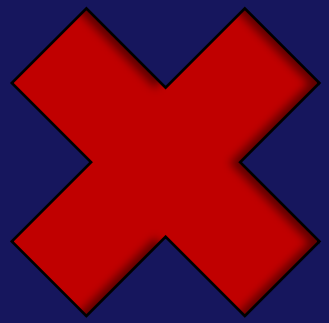
Memory conflict



EURO Greece

Dot product: GPU implementation (3/7)

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



Passing arguments by reference is invalid

EURO Greece

Dot product: GPU implementation (4/7)

```
__global__ void dotProductGPU(const long size, double* _A, double* _B)
{
      const long i = blockDim.x*blockIdx.x + threadIdx.x;
      if (i < size) {
            const double partial_sum = _A[i] * _B[i];
            atomicAdd(&_C,partial_sum);
      }
}

__global__ void dotProductGPU(const long size, double* _A, double* _B)
{
      double partial_sum = 0.0;
      for (long i = blockDim.x*blockIdx.x + threadIdx.x; ; i += gridDim.x*blockDim.x)
      {
            if (i < size) {
                  partial_sum += _A[i] * _B[i];
            }
            else break;
      }
      atomicAdd(&_C,partial_sum);
}
```

EURO
Greece

Dot product: GPU implementation (5/7)

```
__global__ void dotProductGPU(const long size, double* _A, double* _B)
{
      double partial_sum = 0.0;
      for (long i = blockDim.x*blockIdx.x + threadIdx.x; ; i += gridDim.x*blockDim.x)
      {
            if (i < size) {
                  partial_sum += _A[i] * _B[i];
            }
            else break;
      }

      block_reduction(partial_sum);

      if (threadIdx.x == 0) atomicAdd(&_C,partial_sum);
}

__device__ void block_reduction(double& partial_sum)
{
      constexpr int blockSize = 1024;
      if (blockSize != blockDim.x) printf("Invalid block size\n");

      __shared__ double partial_sums[blockSize];
      partial_sums[threadIdx.x] = partial_sum;

      __syncthreads();

      if (threadIdx.x == 0) {
            for (int iThread=1; iThread < blockDim.x; iThread++) partial_sum += partial_sums[iThread];
      }
}
```

Block-wise reduction

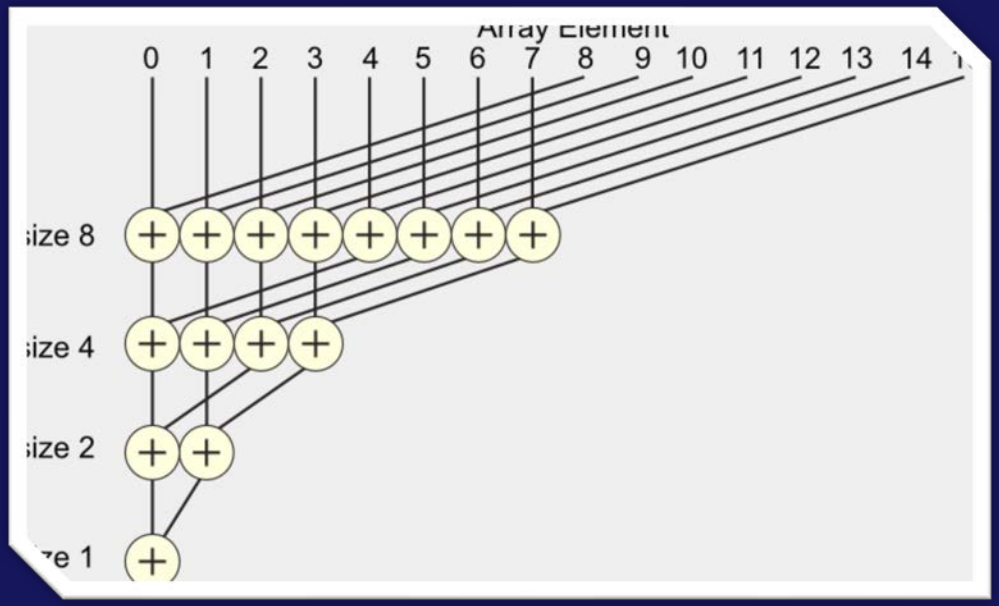
Only the first thread per block performs atomicAdd

Static allocation; the allocation size should be known at compile time

EURO Greece

Dot product: GPU implementation (6/7)

```
__device__ void block_reduction(double& partial_sum)
{
      constexpr int blockSize = 1024;
      if (blockSize != blockDim.x) printf("Invalid block size\n");

      __shared__ double partial_sums[blockSize];
      partial_sums[threadIdx.x] = partial_sum;

      for (int stride=blockDim.x/2; stride > 0; stride /= 2) {
            __syncthreads();
            if (threadIdx.x < stride)
                  partial_sums[threadIdx.x] += partial_sums[threadIdx.x + stride];
      }

      if (threadIdx.x == 0) {
            partial_sum = partial_sums[0];
      }
}
```



EURO
Greece

Dot product: GPU implementation (7/7)

```
#define FULL_MASK 0xffffffff


__device__ void block_reduction(double& partial_sum)
{
      // Warp-wise reduction:
      const int iWarp   = threadIdx.x / warpSize;       // Warp ID within the block
      const int iThread = threadIdx.x - warpSize*iWarp; // Thread ID within the warp

      for (int stride=warpSize/2; stride > 0; stride /= 2) {
            partial_sum += __shfl_down_sync(FULL_MASK, partial_sum, stride);
      }

      // Block-wise reduction:
      const int nbWarps = blockDim.x / warpSize;

      __shared__ double partial_sums[32];
      if (iThread == 0) partial_sums[iWarp] = partial_sum;

      for (int stride=nbWarps/2; stride > 0; stride /= 2) {
            __syncthreads();
            if (iThread == 0 && iWarp < stride) partial_sums[iWarp] += partial_sums[iWarp + stride];
      }

      if (threadIdx.x == 0) {
            partial_sum = partial_sums[0];
      }
}
```

EURO
Greece

Matrix-matrix multiplication

```
class Matrix
{
public:
      Matrix(const int ni, const int nj, const double value = NAN);
      Matrix(const Matrix& matrix);
      ~Matrix();

private:
      int ni_ = 0;
      int nj_ = 0;
      double* values_ = nullptr;

public:
      __host__ void random_init() const;
      __host__ __device__ bool is_valid(const int i, const int j) const;

      __host__ __device__ int ni() const { return ni_; }
      __host__ __device__ int nj() const { return nj_; }

      __host__ __device__ double& operator()(const int i, const int j){ return values_[i*nj_+j]; }
      __host__ __device__ double  operator()(const int i, const int j) const { return values_[i*nj_+j]; }
};
```

```
matrixMultGPU <<< gridSize, blockSize >>> (A,B,C);
```

EURO
Greece

Matrix-matrix multiplication

```
Matrix::Matrix(const int ni, const int nj, const double value)
: ni_(ni), nj_(nj)
{
      values_ = (double*)alloc_managed_mem(ni_*nj_*sizeof(double), "Matrix::Elements");
      for (int index=0; index < ni_*nj_; index++) values_[index] = value;
}

Matrix::Matrix(const Matrix& matrix)
{
      ni_     = matrix.ni_;
      nj_     = matrix.nj_;
      values_ = matrix.values_;
}

Matrix::~Matrix()
{
      cudaFree(values_);
}
```

Allocate managed memory

Shallow copy

Free memory by using cudaFree

EURO Greece

Matrix-matrix multiplication

```
int main()
{
      constexpr int NI = 150;
      constexpr int NJ = 200;
      constexpr int NK = 120;

      Matrix A(NI,NJ), B(NJ,NK), C(NI,NK,0.0);

      A.random_init();
      B.random_init();

      for (int i=0; i < C.ni(); i++)
      for (int j=0; j < C.nj(); j++)
      {
            for (int k=0; k < A.nj(); k++)
            {
                  C(i,j) += A(i,k)*B(k,j);
            }
      }

      return 0;
}
```

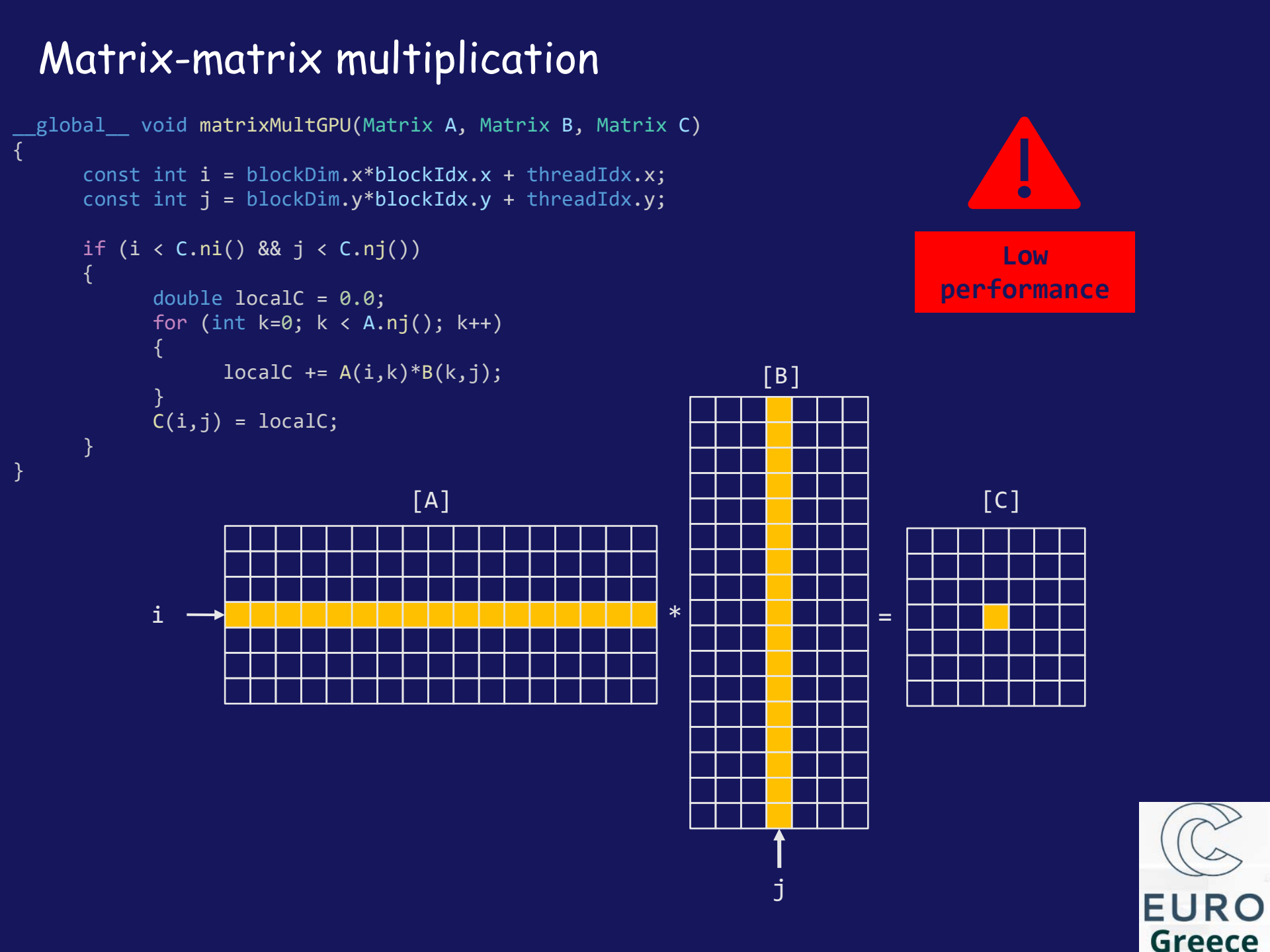
Matrix-matrix multiplication

```
int main()
{
      constexpr int NI = 150;
      constexpr int NJ = 200;
      constexpr int NK = 120;

      Matrix A(NI,NJ), B(NJ,NK), C(NI,NK,0.0);

      A.random_init();
      B.random_init();

      dim3 gridSize, blockSize = {32,32};
      gridSize.x = (C.ni()+blockSize.x-1)/blockSize.x;
      gridSize.y = (C.nj()+blockSize.y-1)/blockSize.y;
      matrixMultGPU <<< gridSize, blockSize >>> (A,B,C);

      cudaDeviceSynchronize();
      cudaError_t err = cudaGetLastError();
      if (err != cudaSuccess) {
            std::cerr << " Launch failure: " << cudaGetErrorString(err) << std::endl;
            exit(1);
      }

      return 0;
}
```

EURO
Greece

Matrix-matrix multiplication

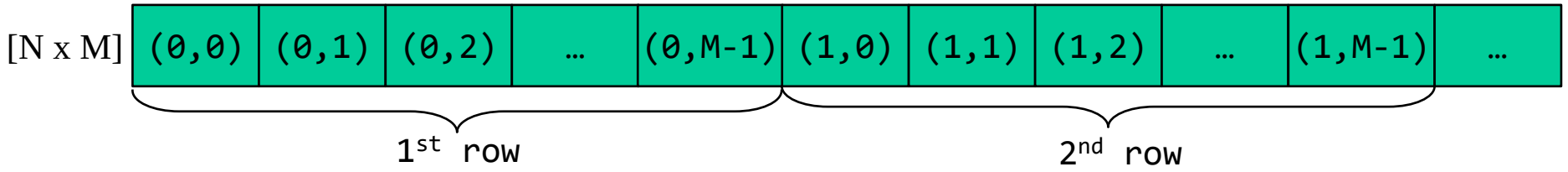
```
__global__ void matrixMultGPU(Matrix A, Matrix B, Matrix C)
{
      const int i = blockDim.x*blockIdx.x + threadIdx.x;
      const int j = blockDim.y*blockIdx.y + threadIdx.y;

      if (i < C.ni() && j < C.nj())
      {
            double localC = 0.0;
            for (int k=0; k < A.nj(); k++)
            {
                  localC += A(i,k)*B(k,j);
            }
            C(i,j) = localC;
      }
}
```



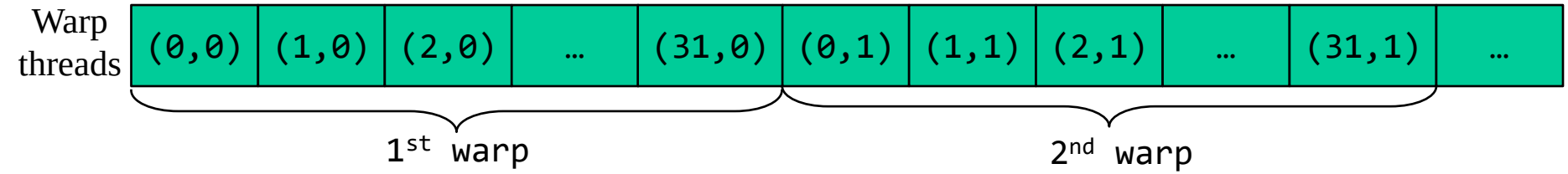
EURO Greece

Matrix-matrix multiplication

Row-major order: index ← i*M + j

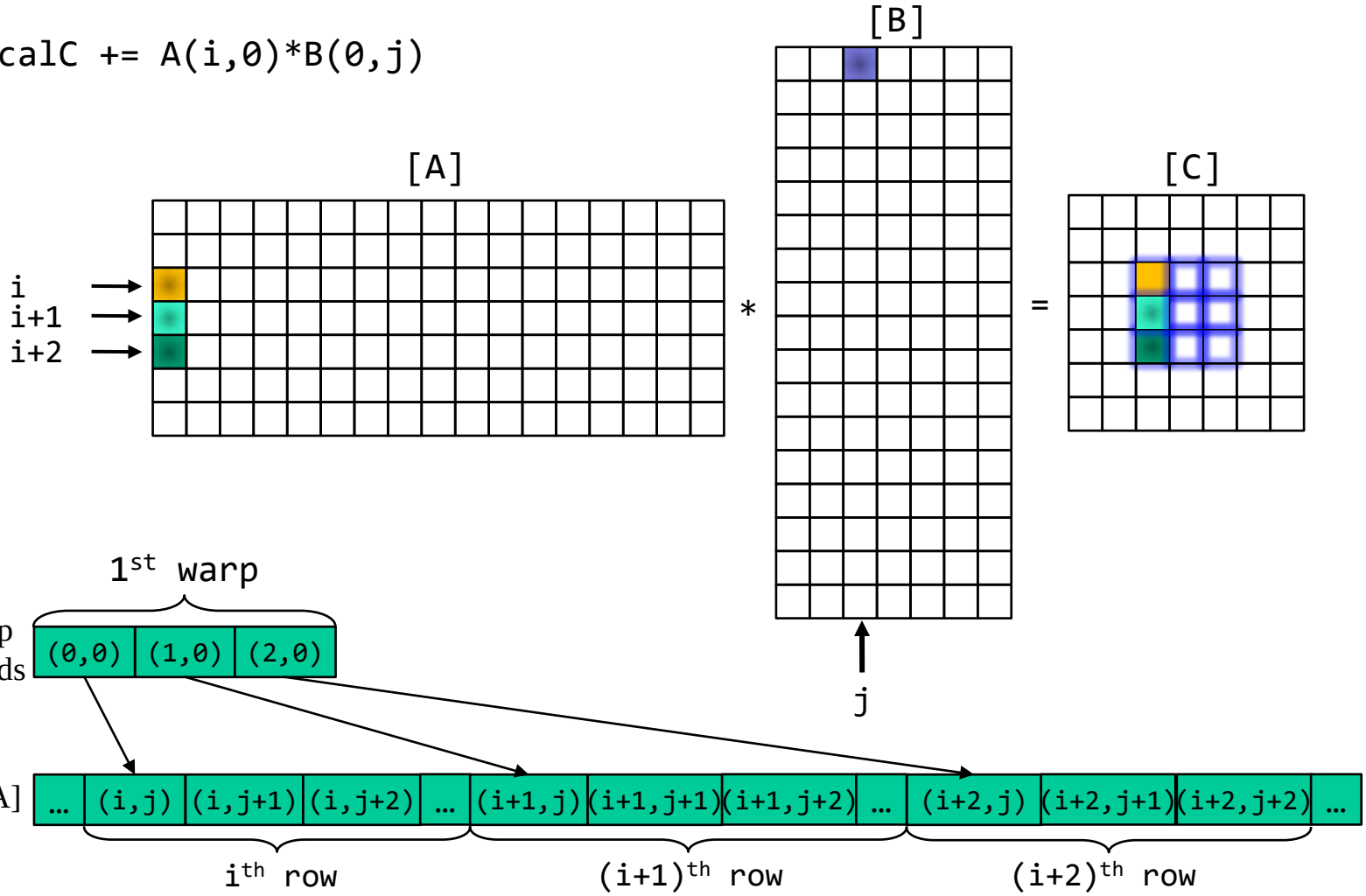


column-major order: iThread ← threadIdx.y*blockDim.y + threadIdx.x



Matrix-matrix multiplication

```
localC += A(i,0)*B(0,j)
```

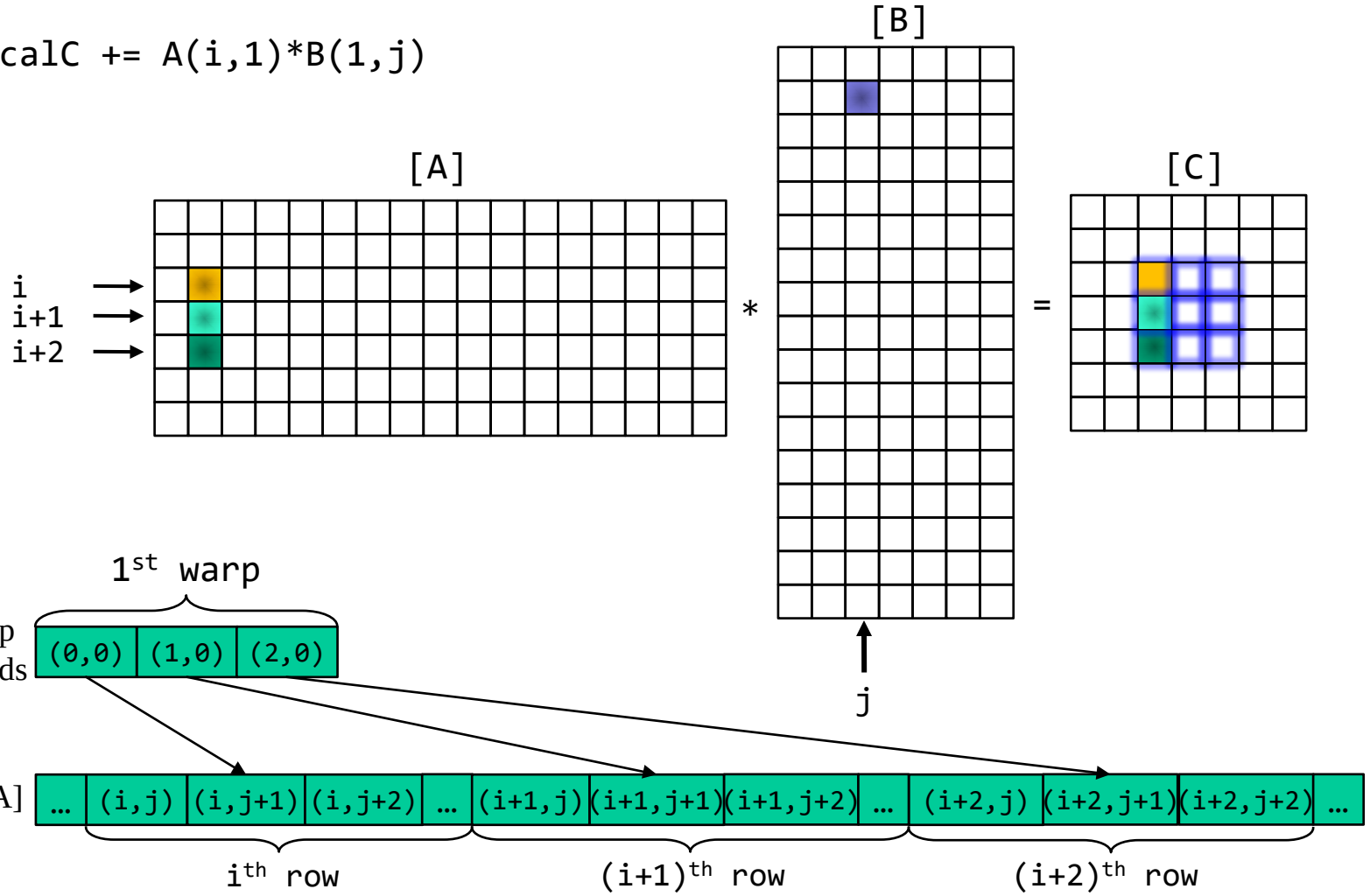



```
i = blockDim.x*blockIdx.x + threadIdx.x;
j = blockDim.y*blockIdx.y + threadIdx.y;
```

EURO Greece

Matrix-matrix multiplication

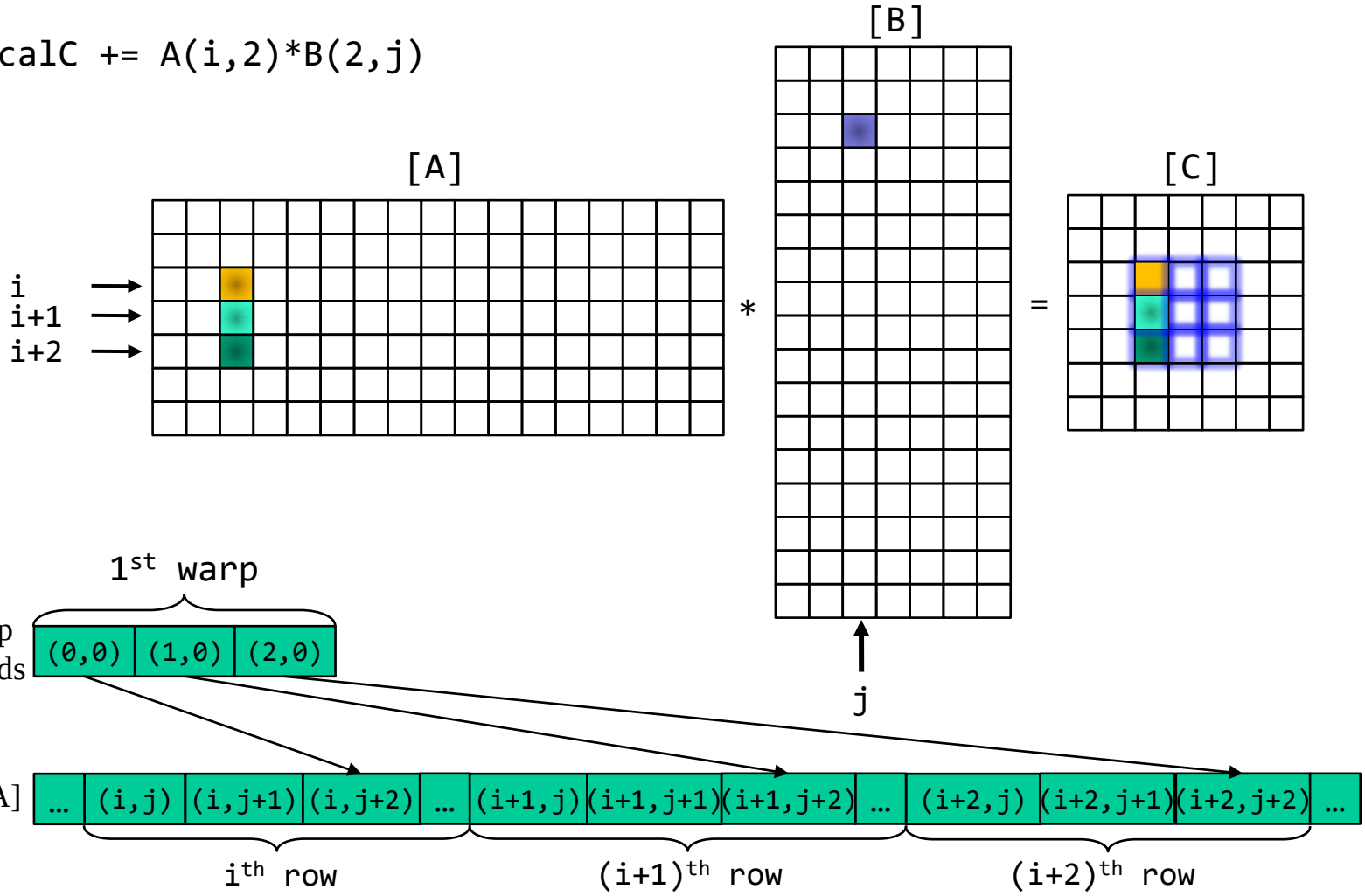
```
localC += A(i,1)*B(1,j)
```




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

Matrix-matrix multiplication

```
localC += A(i,2)*B(2,j)
```




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

EURO Greece

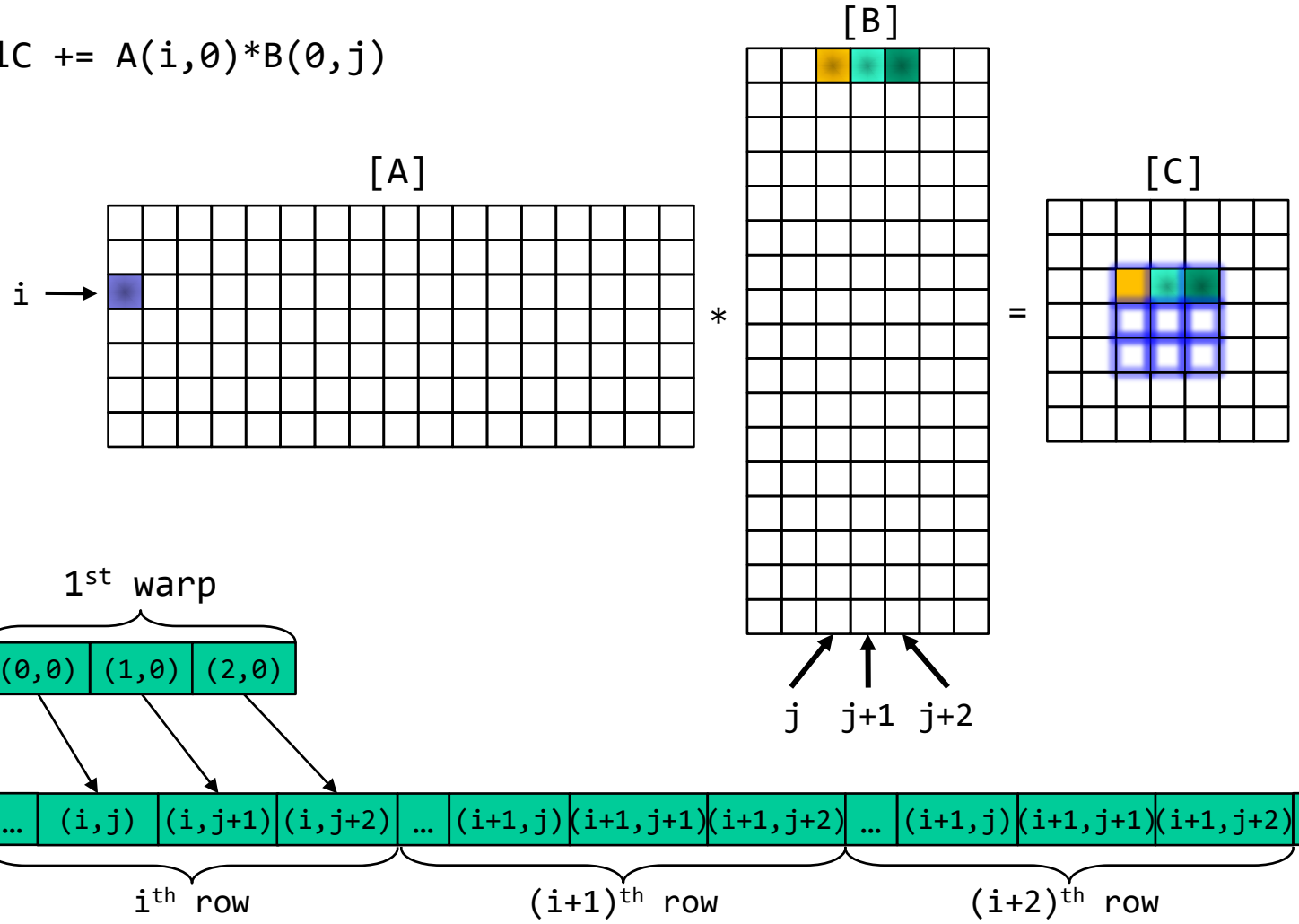
Matrix-matrix multiplication

```
gridSize.x = (C.nj()+blockSize.x-1)/blockSize.x;
gridSize.y = (C.ni()+blockSize.y-1)/blockSize.y;


__global__ void matrixMultGPU(Matrix A, Matrix B, Matrix C)
{
      const int j = blockDim.x*blockIdx.x + threadIdx.x;
      const int i = blockDim.y*blockIdx.y + threadIdx.y;

      if (i < C.ni() && j < C.nj())
      {
            double localC = 0.0;
            for (int k=0; k < A.nj(); k++)
            {
                  localC += A(i,k)*B(k,j);
            }
            C(i,j) = localC;
      }
}
```

EURO
Greece

Matrix-matrix multiplication

```
localC += A(i,0)*B(0,j)
```

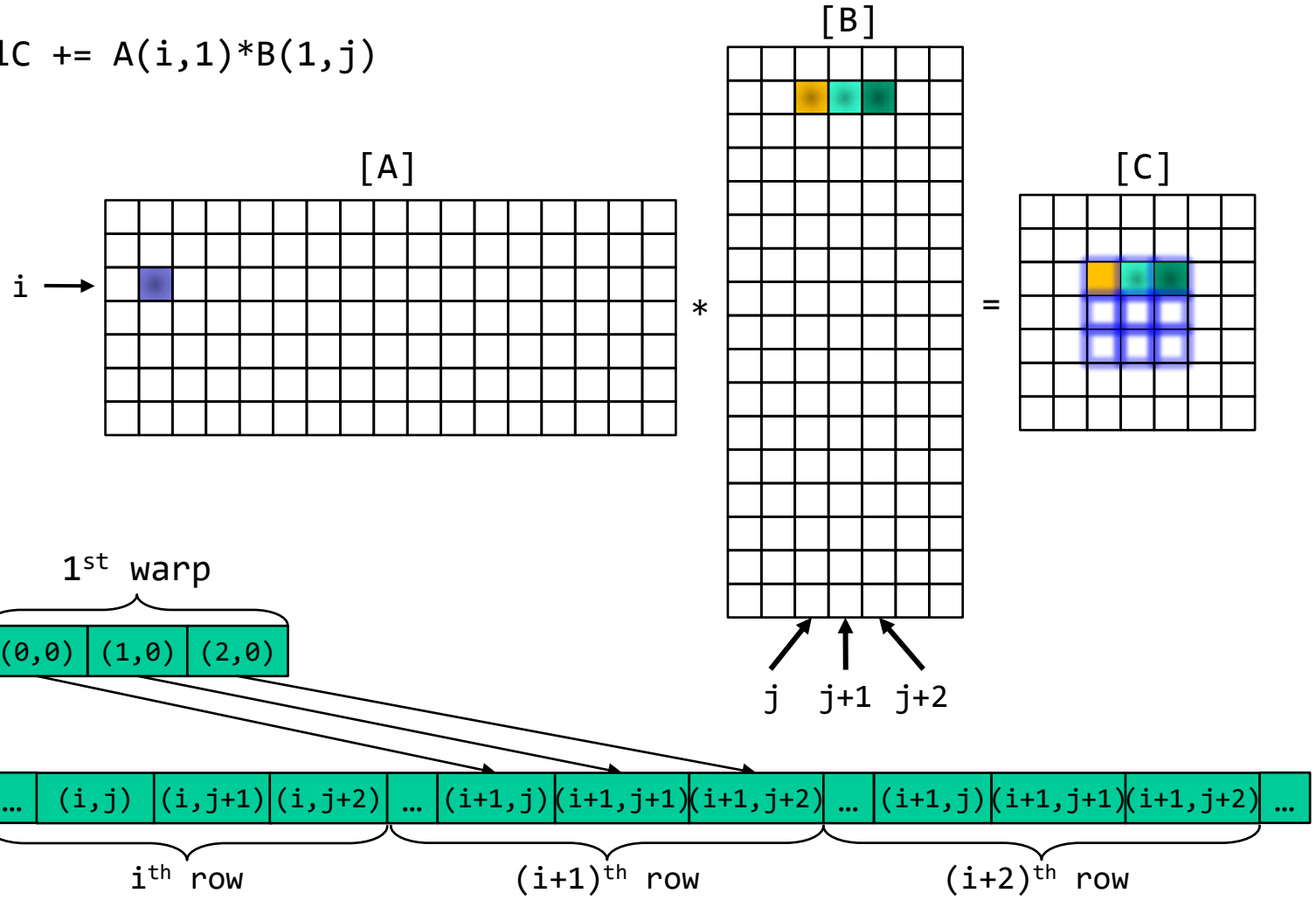



```
j = blockDim.x*blockIdx.x + threadIdx.x;
i = blockDim.y*blockIdx.y + threadIdx.y;
```

EURO Greece

Matrix-matrix multiplication

```
localC += A(i,1)*B(1,j)
```

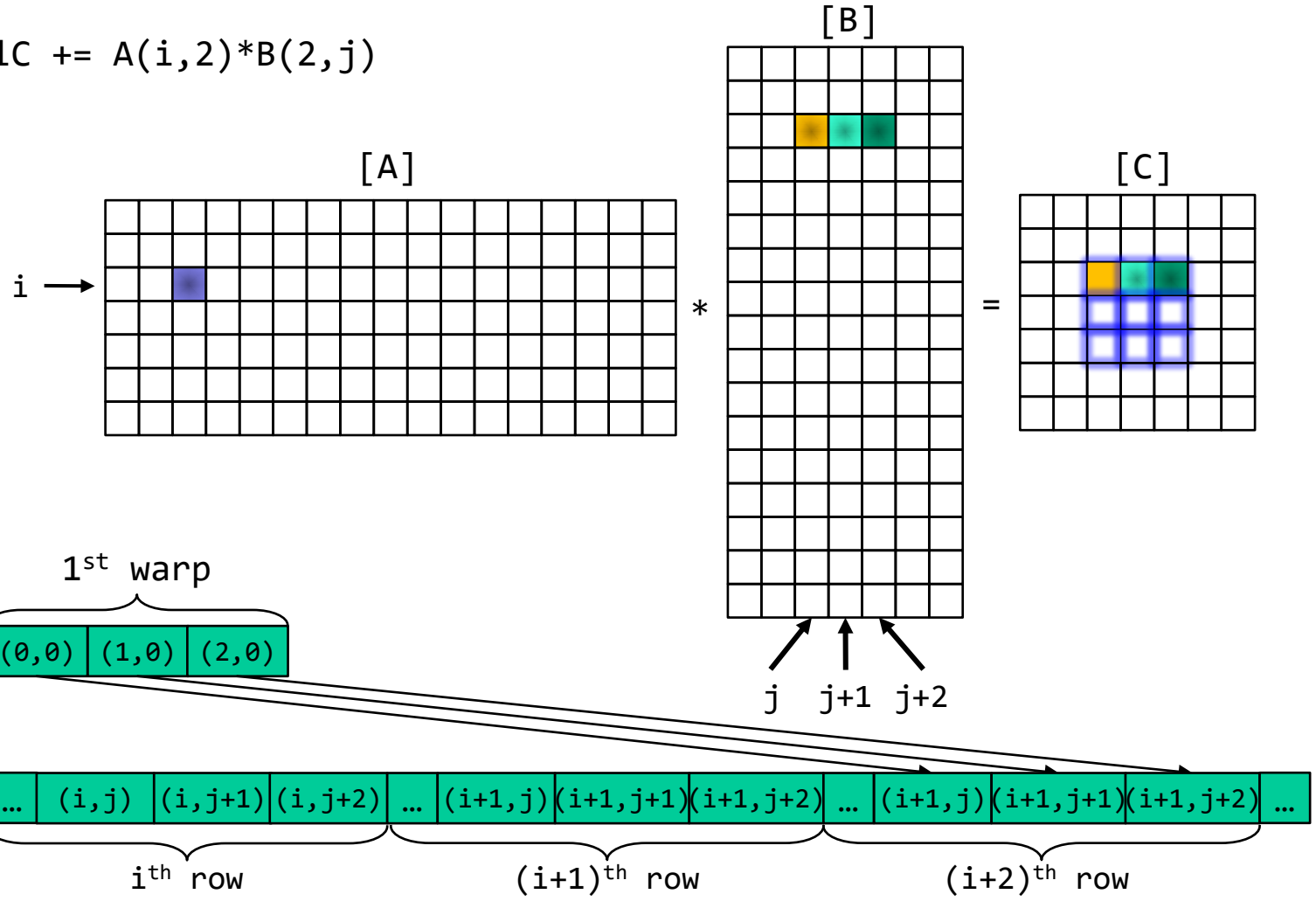



```
j = blockDim.x*blockIdx.x + threadIdx.x;
i = blockDim.y*blockIdx.y + threadIdx.y;
```

EURO Greece

Matrix-matrix multiplication

```
localC += A(i,2)*B(2,j)
```

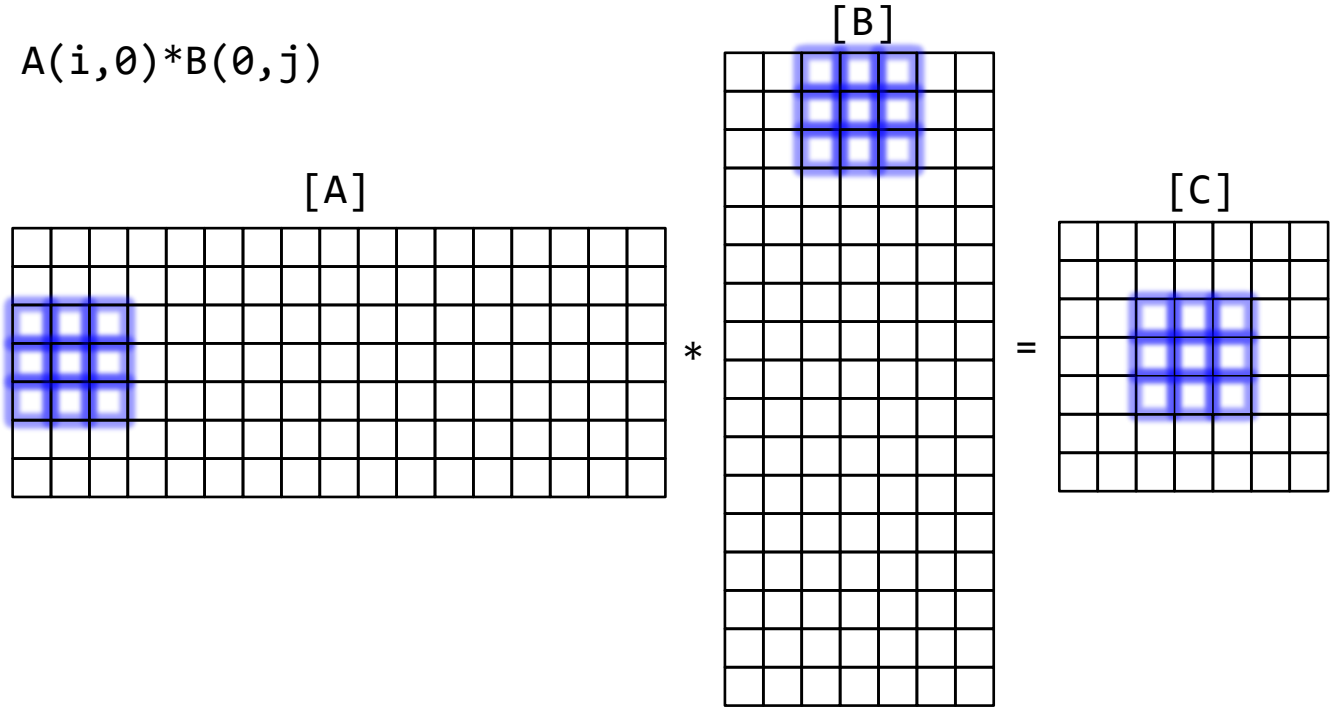



```
j = blockDim.x*blockIdx.x + threadIdx.x;
i = blockDim.y*blockIdx.y + threadIdx.y;
```

EURO Greece

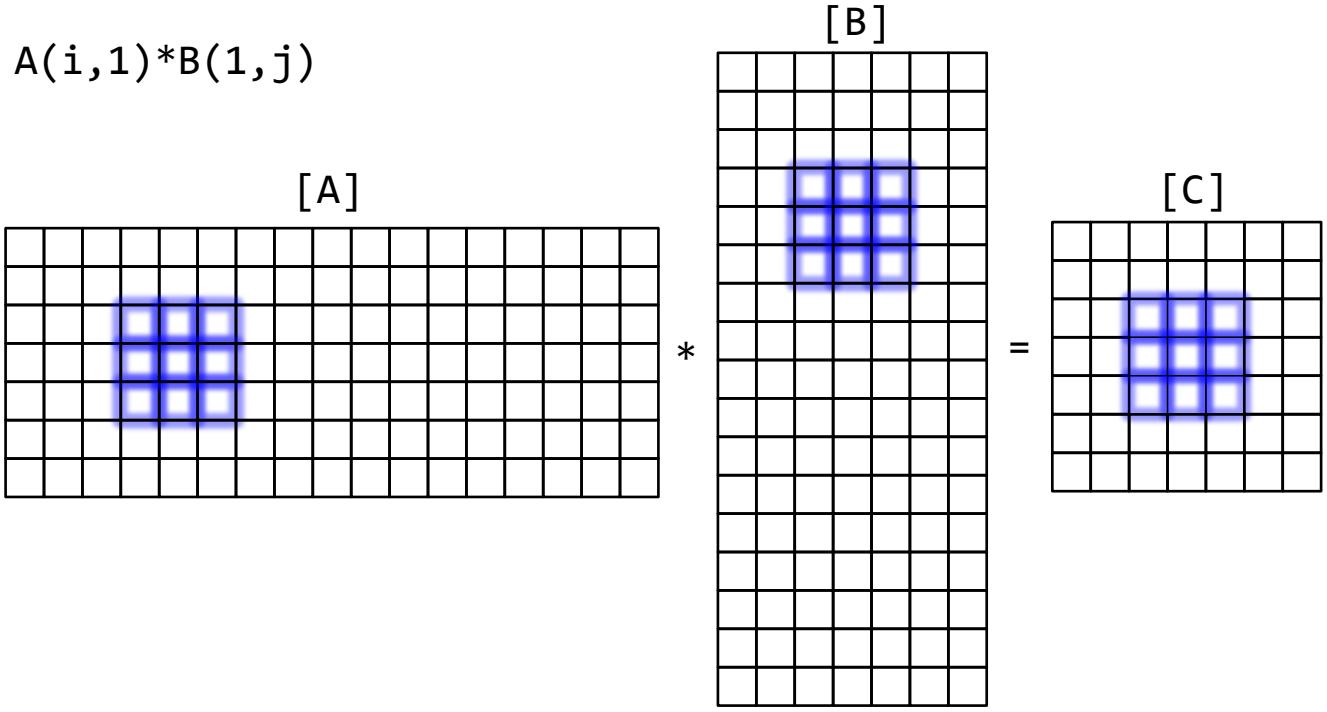
Matrix-matrix multiplication

```
localC += A(i,0)*B(0,j)
```



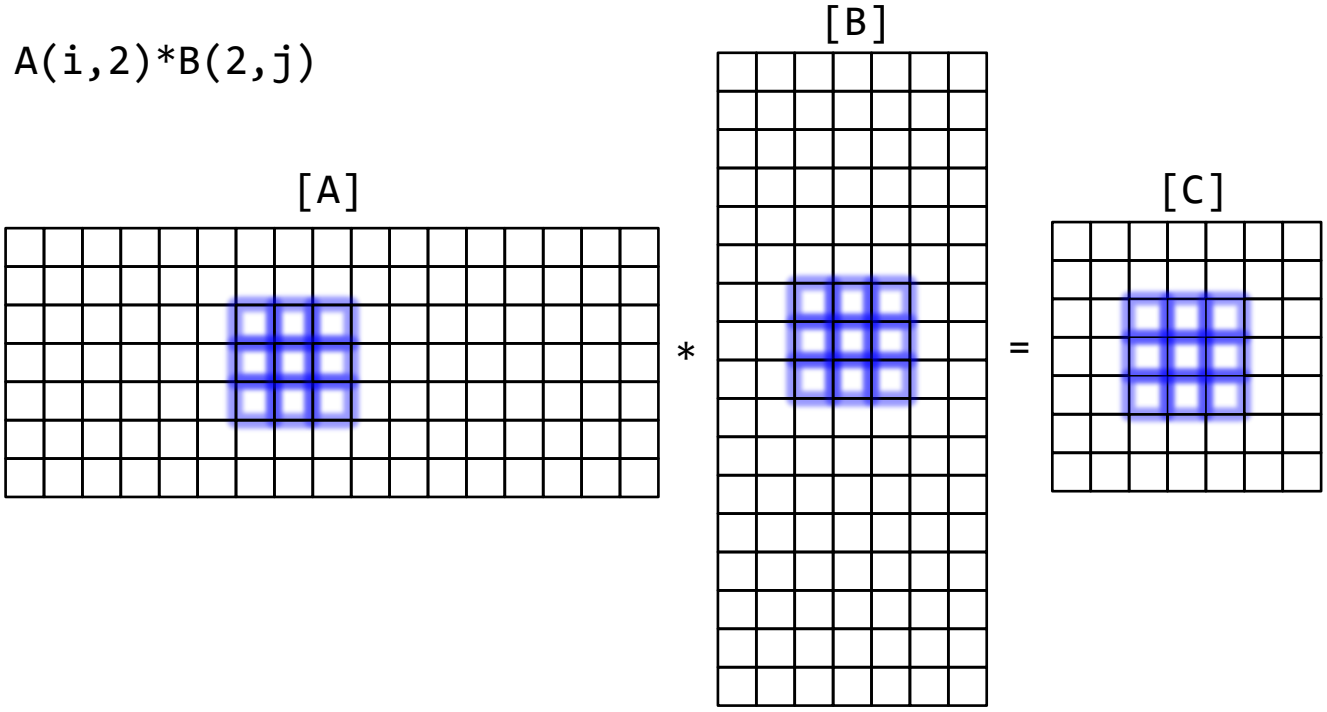
Matrix-matrix multiplication

```
localC += A(i,1)*B(1,j)
```



Matrix-matrix multiplication

```
localC += A(i,2)*B(2,j)
```



Matrix-matrix multiplication

```
__global__ void matrixMultGPU(Matrix A, Matrix B, Matrix C)
{
      const int j = blockDim.x*blockIdx.x + threadIdx.x;
      const int i = blockDim.y*blockIdx.y + threadIdx.y;

      constexpr short ts = 32;
      __shared__ double subA[ts][ts], subB[ts][ts];

      const int nbTiles = (A.nj()+ts-1)/ts;

      double localC = 0.0;
      for (int iTile=0; iTile < nbTiles; iTile++)
      {
            subA[threadIdx.y][threadIdx.x] = 0.0;
            subB[threadIdx.y][threadIdx.x] = 0.0;

            const int k = iTile*ts;
            if (A.is_valid(i,k+threadIdx.x) == true) subA[threadIdx.y][threadIdx.x] = A(i,k+threadIdx.x);
            if (B.is_valid(k+threadIdx.y,j) == true) subB[threadIdx.y][threadIdx.x] = B(k+threadIdx.y,j);
            __syncthreads();

            for (int l=0; l<ts; l++) {
                  localC += subA[threadIdx.y][l] * subB[l][threadIdx.x];
            }
            __syncthreads();
      }

      if (C.is_valid(i,j)) C(i,j) = localC;
}
```

The barrier ensures that all block threads load submatrices of A and B into shared memory

The barrier ensures that no thread re-writes on shared memory until all threads have updated variable *localC*

*** More optimizations …

Non-included items

- Multi-GPUs; same host controlling many GPUs connected to the same CPU via PCIe
- Pinned memory; enables threads to access host memory
- Cooperative groups; enable grid-level or even multigrid-level synchronizations
- Dynamic parallelism; enables kernel launches within kernels
- CUDA streams & events; enable kernel launches and memory copies to overlap
- Constant & Texture memory; Utilizes constant and texture caches, texture filtering and access models
- Tensor cores; cuBLAS & cuDNN libraries, TensorFlow & pyTorch deep-learning frameworks
- Libraries; cuBLAS, cuSPARCE, cuSOLVER, cuDNN, cuFFT, AMGX
- …

EURO Greece