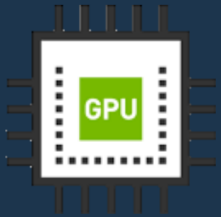
ESCAPE 824064

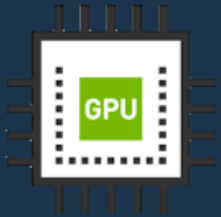
GPU Computing with C++20 nvc++ and CUDA

Pierre Aubert

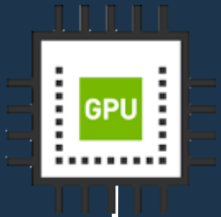
Standard C++ for CPU and GPU



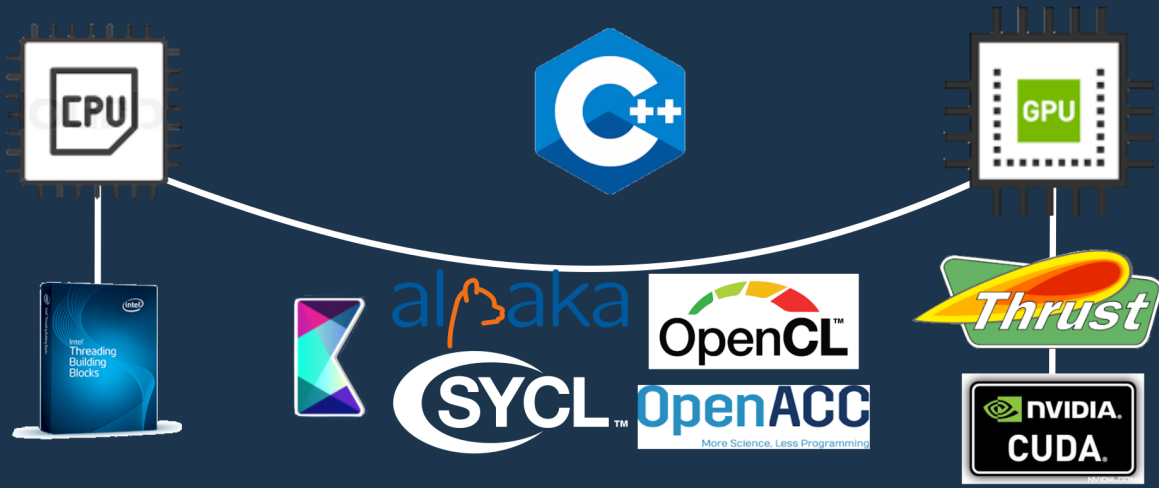
Standard C++ for CPU and GPU



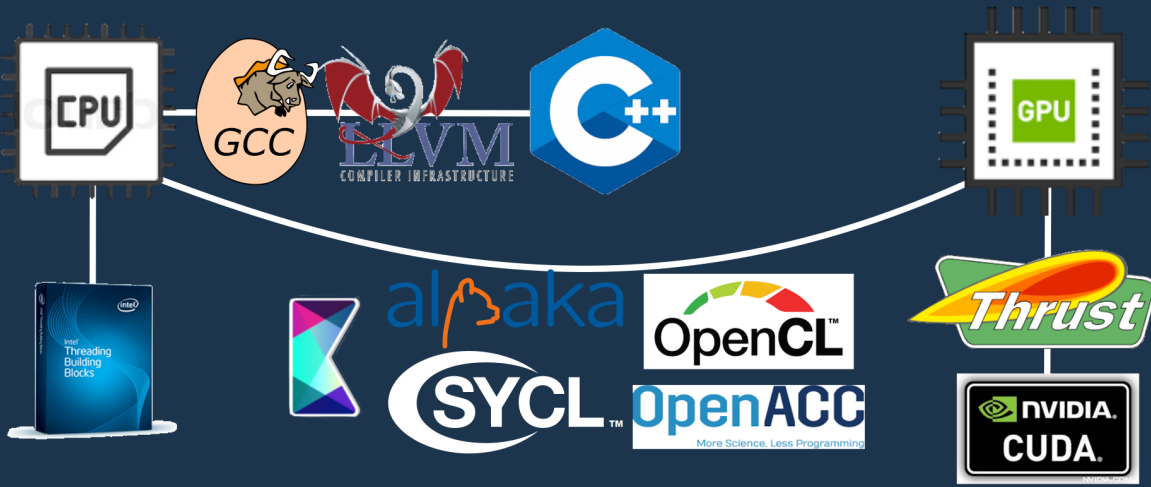
Standard C++ for CPU and GPU



Standard C++ for CPU and GPU

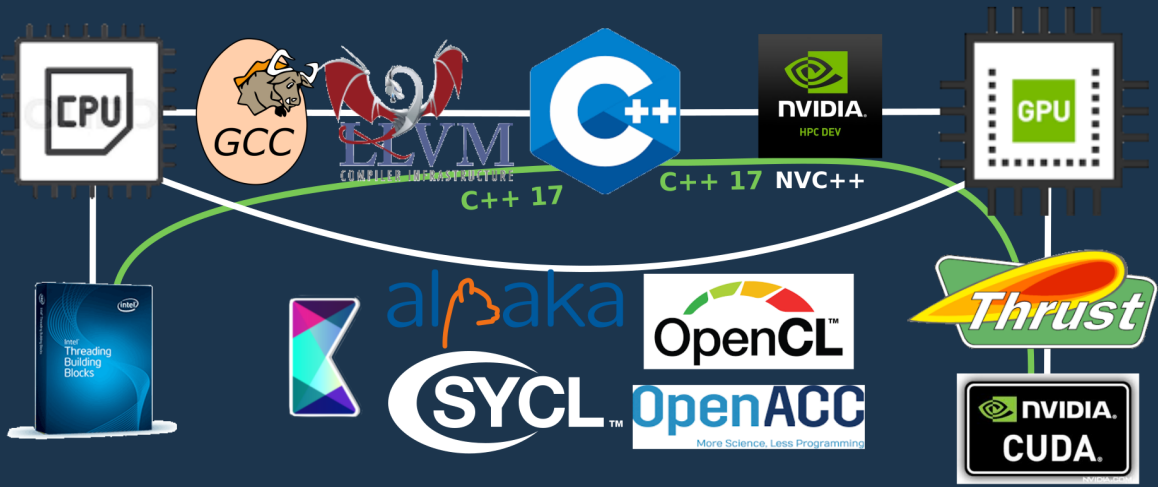


Standard C++ for CPU and GPU



Standard C++ for CPU and GPU

Standard C++ for CPU and GPU



Express Parallelism

Express Parallelism

Event

Express Parallelism

Event

Independent Events => Independent Computing => Parallelism

Express Parallelism

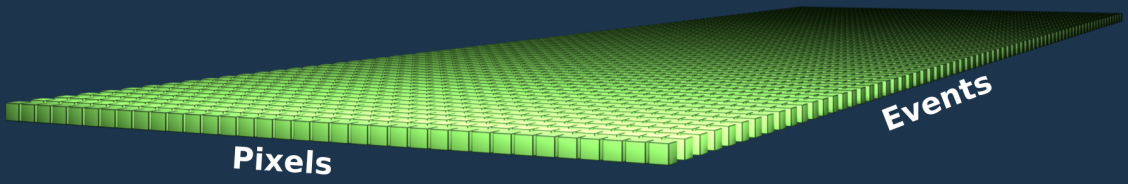
Event

Independent Events => Independent Computing => Parallelism

Express Parallelism

Event

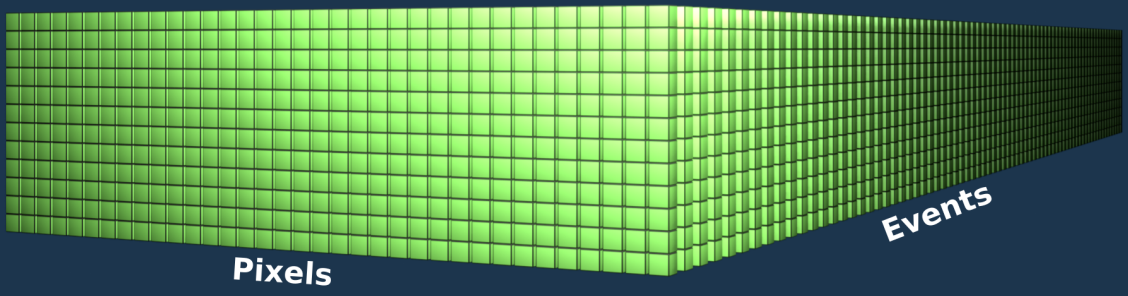
Independent Events => Independent Computing => Parallelism



Express Parallelism

Event

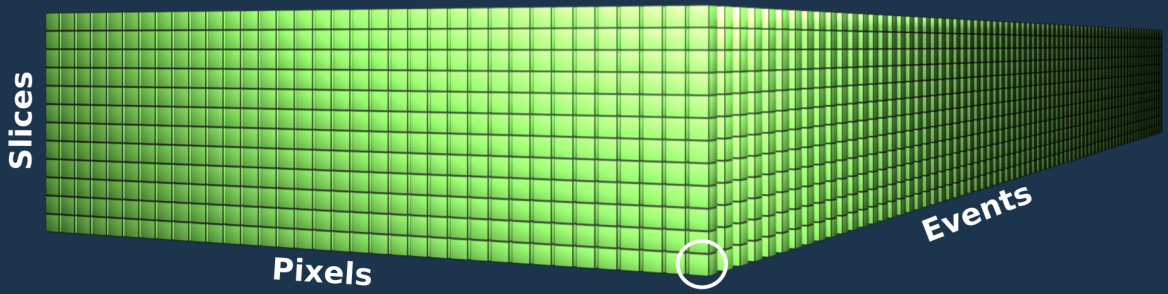
Independent Events => Independent Computing => Parallelism



Express Parallelism

Event

Independent Events => Independent Computing => Parallelism

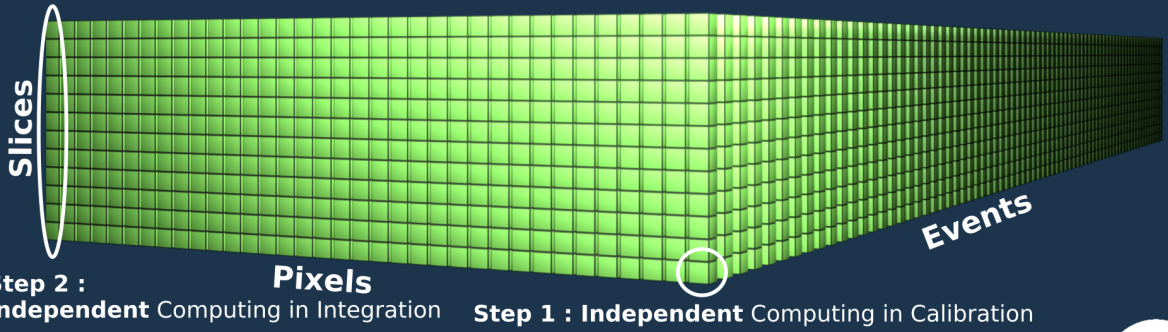


Step 1 : Independent Computing in Calibration

Express Parallelism

Event

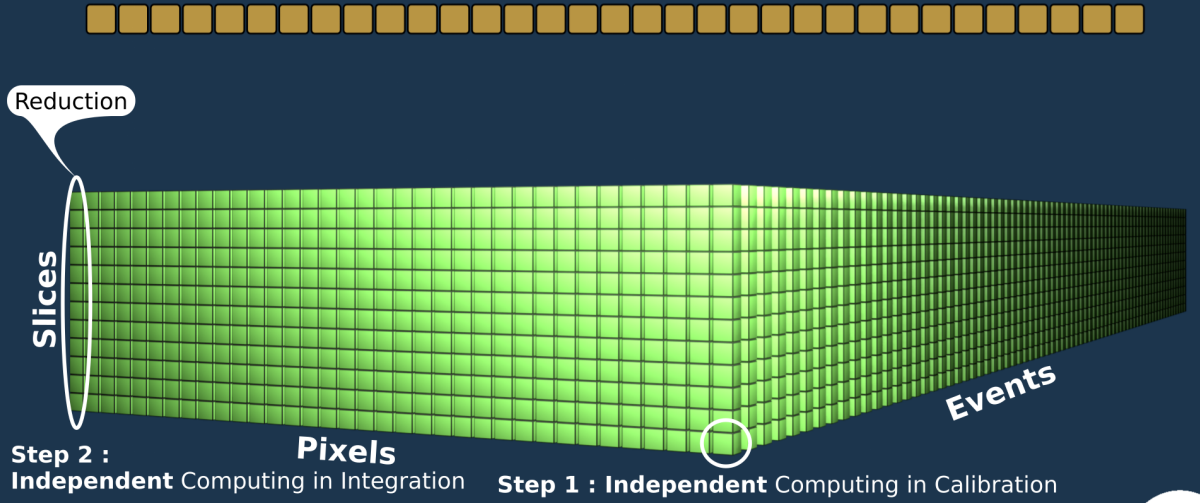
Independent Events => Independent Computing => Parallelism



Express Parallelism

Event

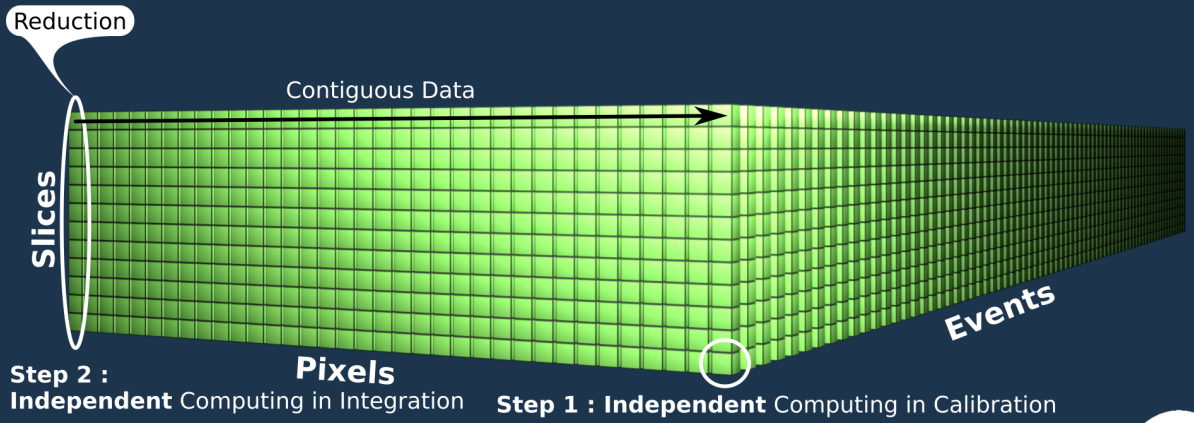
Independent Events => Independent Computing => Parallelism



Express Parallelism

Event

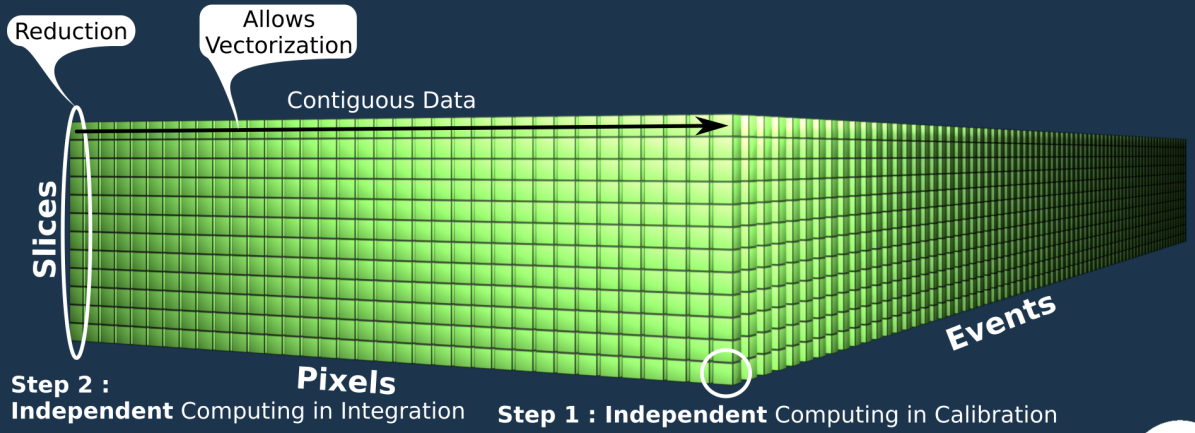
Independent Events => Independent Computing => Parallelism



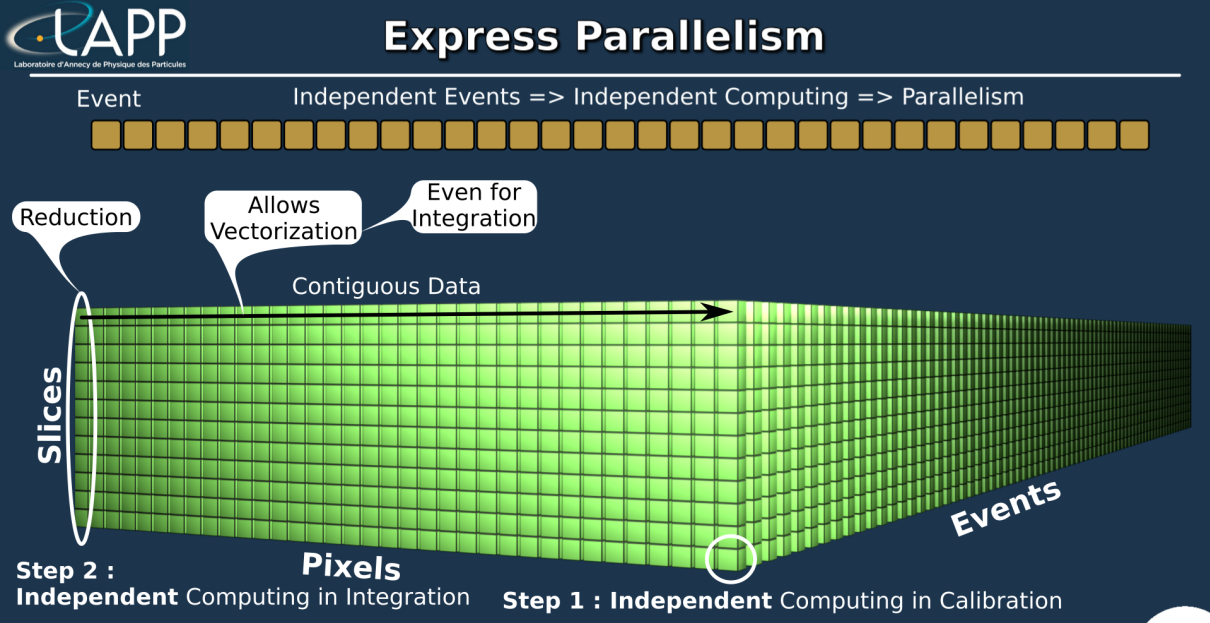
Express Parallelism

Event

Independent Events => Independent Computing => Parallelism



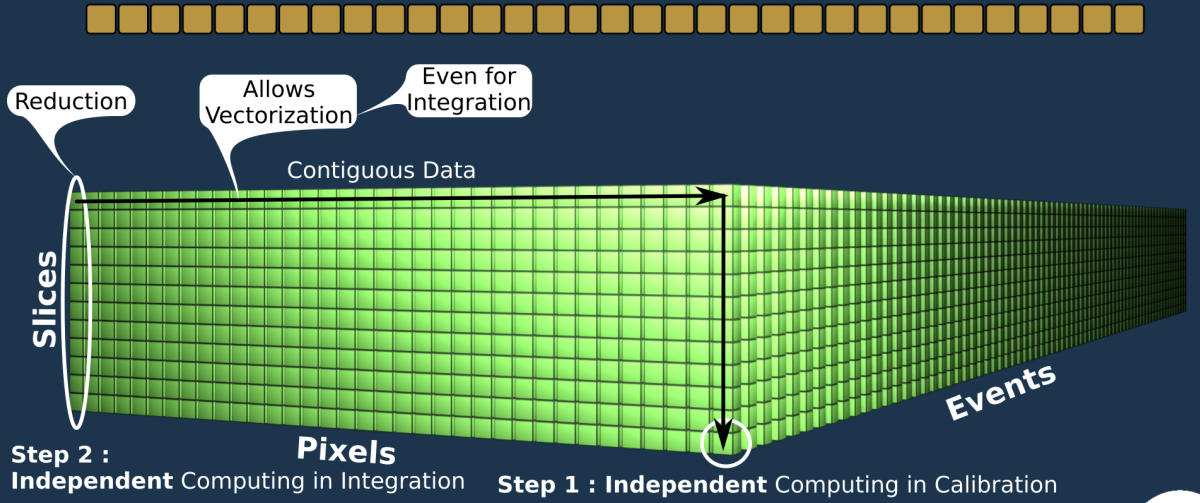
Express Parallelism



Express Parallelism

Event

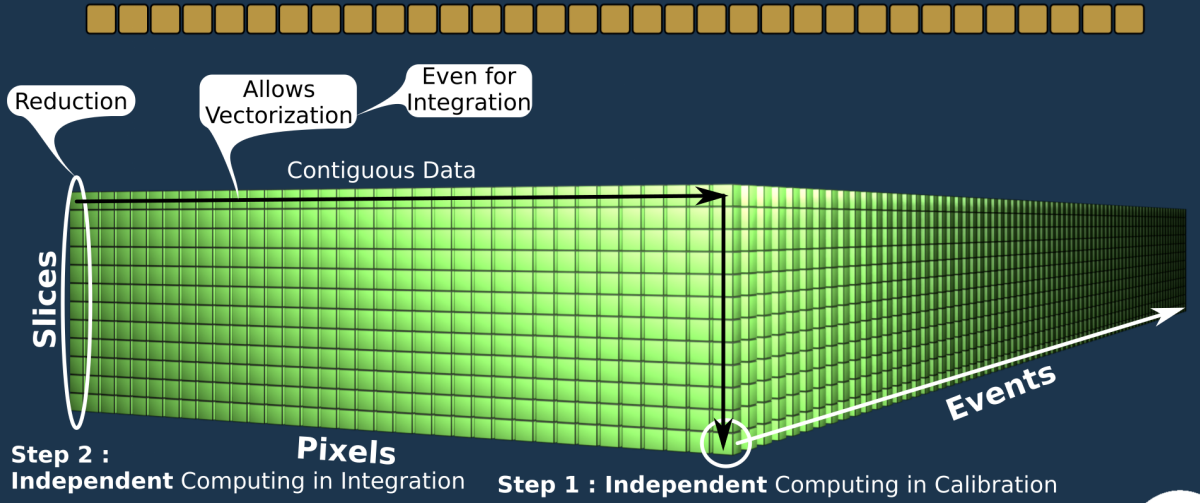
Independent Events => Independent Computing => Parallelism



Express Parallelism

Event

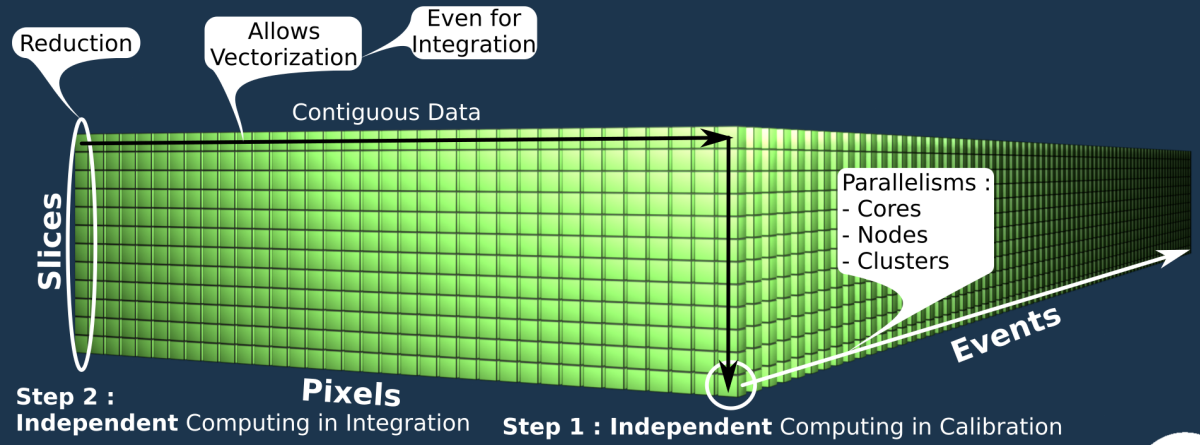
Independent Events => Independent Computing => Parallelism



Express Parallelism

Event

Independent Events => Independent Computing => Parallelism



Express Parallelism

Event

Independent Events => Independent Computing => Parallelism

Reduction

Allows Vectorization

Even for Integration

Computing Drives Data Storage

Contiguous Data

Slices

Pixels

Parallelisms :
- Cores
- Nodes
- Clusters

Events

Step 2 : Independent Computing in Integration

Step 1 : Independent Computing in Calibration

The CUDA ecosystem

The CUDA ecosystem

Kernel Development

CUDA **cu++** **CUTLASS**

The CUDA ecosystem

Computing Libraries

BLAS | LAPACK | PBLAS | SCALAPACK

TENSOR | SPARSE | RAND | FFT

Kernel Development

CUDA | cu++ | CUTLASS

The CUDA ecosystem

Computing Libraries

BLAS | LAPACK | PBLAS | SCALAPACK

TENSOR | SPARSE | RAND | FFT

Kernel Development

CUDA | cu++ | CUTLASS

The CUDA ecosystem

Computing Libraries

BLAS | LAPACK | PBLAS | SCALAPACK

TENSOR | SPARSE | RAND | FFT

Thrust

Computing + I/O

RAPIDS

cunumeric | legate | cuDF

Kernel Development

CUDA | cu++ | CUTLASS

NVIDIA CUDA

The CUDA ecosystem

Laboratoire d'Annecy de Physique des Particules

Computing Libraries

BLAS | LAPACK | PBLAS | SCALAPACK

TENSOR | SPARSE | RAND | FFT

Thrust

Computing + I/O

RAPIDS

cunumeric | legate | cuDF

Kernel Development

CUDA | cu++ | CUTLASS

NVIDIA CUDA

I/O and communication

MagnumIO

NVShmem | GPUDirect | NCCL

The CUDA ecosystem

Laboratoire d'Annecy de Physique des Particules

Computing Libraries

BLAS | LAPACK | PBLAS | SCALAPACK

TENSOR | SPARSE | RAND | FFT

Thrust

Computing + I/O

RAPIDS

cunumeric | legate | cuDF

Kernel Development

CUDA | cu++ | CUTLASS

NVIDIA CUDA

Profiler / Debugger

NSight

I/O and communication

MagnumIO

NVShmem | GPUDirect | NCCL

Computing on GPU with CUDA

Express Global Computation -> Linear Algebra

Computing on GPU with CUDA

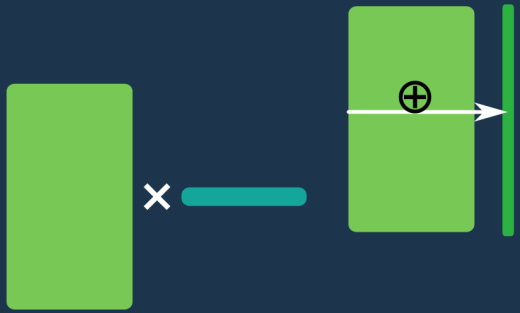
Express Global Computation -> Linear Algebra

Calibration -> Broadcast

Computing on GPU with CUDA

Express Global Computation -> Linear Algebra

Calibration -> Broadcast **Signal Reduction -> SGEMV**

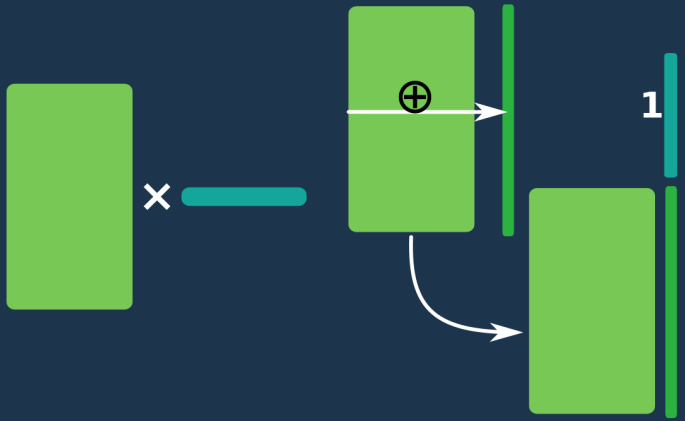


Computing on GPU with CUDA

Express Global Computation -> Linear Algebra

Calibration -> Broadcast

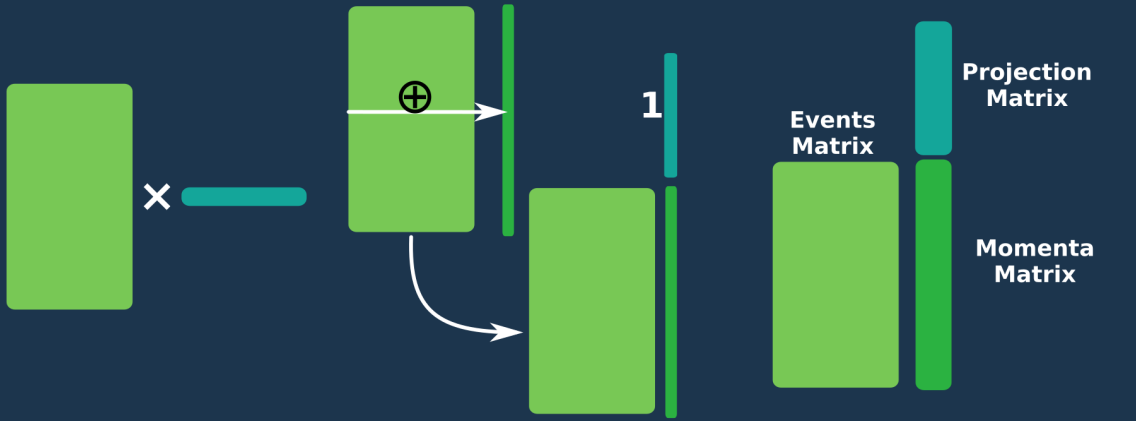
Signal Reduction -> SGEMV



Computing on GPU with CUDA

Express Global Computation -> Linear Algebra

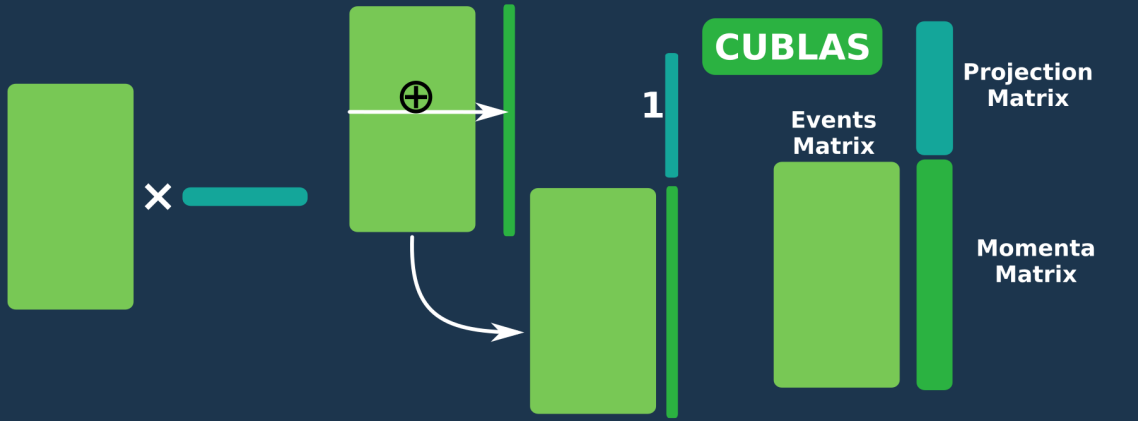
Calibration -> Broadcast | Signal Reduction -> SGEMV | Momenta computation -> SGEMM



Computing on GPU with CUDA

Express Global Computation -> Linear Algebra

Calibration -> Broadcast | Signal Reduction -> SGEMV | Momenta computation -> SGEMM



Threads Hierarchy

Threads Hierarchy

Thread

Threads Hierarchy

Thread Core

Threads Hierarchy

Thread Core Executes Computing Kernel

Threads Hierarchy

Warp

Thread Core Executes Computing Kernel

Threads Hierarchy

Warp Hardware Scheduler

Thread Core Executes Computing Kernel

Threads Hierarchy

Warp Hardware Scheduler 32 **Threads**

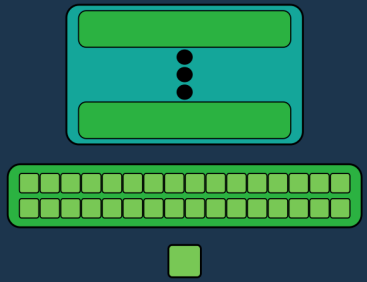
Thread Core Executes Computing Kernel

Threads Hierarchy

Warp Hardware Scheduler 32 **Threads** **Synchronous** Same Instruction Multiple Data

Thread Core Executes Computing Kernel

Threads Hierarchy

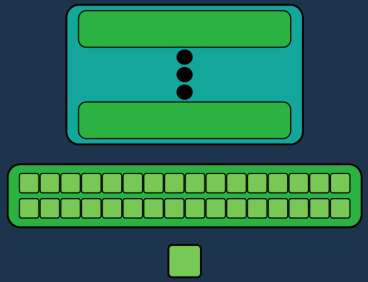


Block

Warp Hardware Scheduler 32 **Threads** **Synchronous** Same Instruction Multiple Data

Thread Core Executes Computing Kernel

Threads Hierarchy

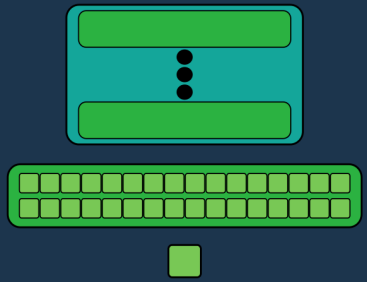


Block Streaming Multiprocessor

Warp Hardware Scheduler 32 **Threads** **Synchronous** Same Instruction Multiple Data

Thread Core Executes Computing Kernel

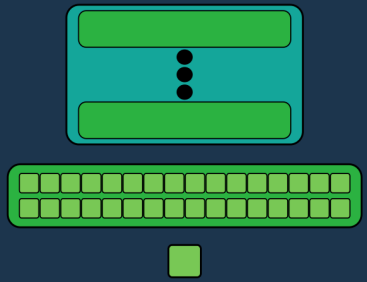
Threads Hierarchy



Block	Streaming Multiprocessor	2 - 4 **Warps** 64 - 128 **Cores**
Warp	Hardware Scheduler	32 **Threads**
Thread	Core	Executes Computing Kernel

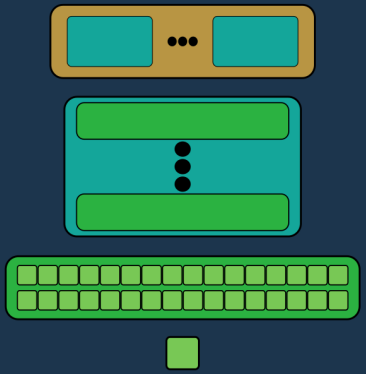
Synchronous
Same Instruction
Multiple Data

Threads Hierarchy



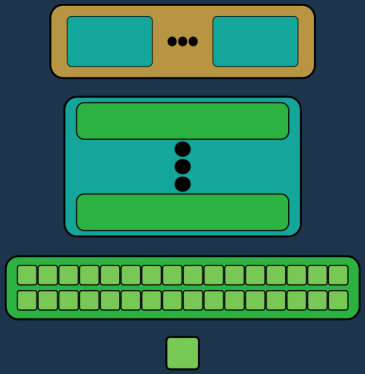
Block	Streaming Multiprocessor	2 - 4 **Warps** 64 - 128 **Cores**	**Asynchronous** Same Instruction Multiple Data
Warp	Hardware Scheduler	32 **Threads**	**Synchronous** Same Instruction Multiple Data
Thread	Core	Executes Computing Kernel	

Threads Hierarchy



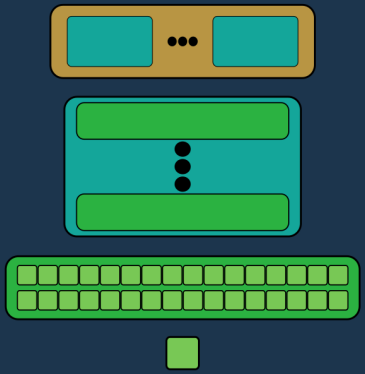
Cluster			
Block	Streaming Multiprocessor	2 - 4 **Warps** 64 - 128 **Cores**	**Asynchronous** Same Instruction Multiple Data
Warp	Hardware Scheduler	32 **Threads**	**Synchronous** Same Instruction Multiple Data
Thread	Core	Executes Computing Kernel	

Threads Hierarchy



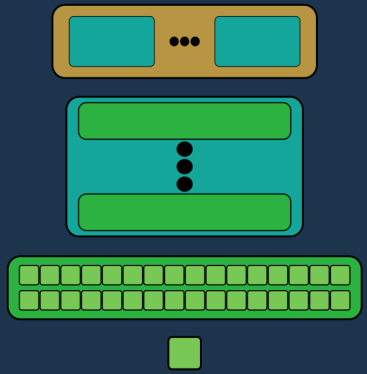
Cluster	Blocks Communication		
Block	Streaming Multiprocessor	2 - 4 **Warps** 64 - 128 **Cores**	**Asynchronous** Same Instruction Multiple Data
Warp	Hardware Scheduler	32 **Threads**	**Synchronous** Same Instruction Multiple Data
Thread	Core	Executes Computing Kernel	

Threads Hierarchy



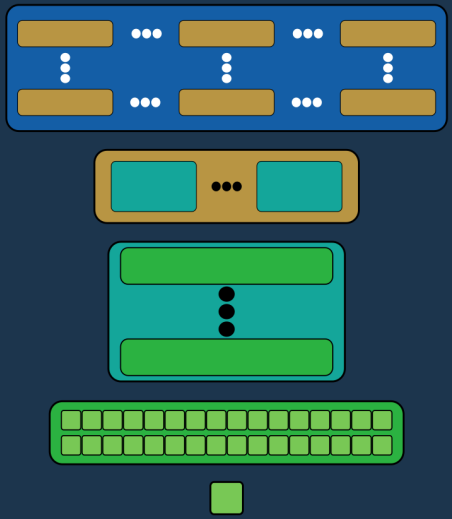
Cluster	Blocks Communication	Since : - CUDA **12** - **Hopper**	
Block	Streaming Multiprocessor	2 - 4 **Warps** 64 - 128 **Cores**	**Asynchronous** Same Instruction Multiple Data
Warp	Hardware Scheduler	32 **Threads**	**Synchronous** Same Instruction Multiple Data
Thread	Core	Executes Computing Kernel	

Threads Hierarchy



Cluster	Blocks Communication	Since : - CUDA **12** - **Hopper**	**Hopper** : 132 SMs **Ampere** : 108 SMs
Block	Streaming Multiprocessor	2 - 4 **Warps** 64 - 128 **Cores**	**Asynchronous** Same Instruction Multiple Data
Warp	Hardware Scheduler	32 **Threads**	**Synchronous** Same Instruction Multiple Data
Thread	Core	Executes Computing Kernel	

Threads Hierarchy



Grid

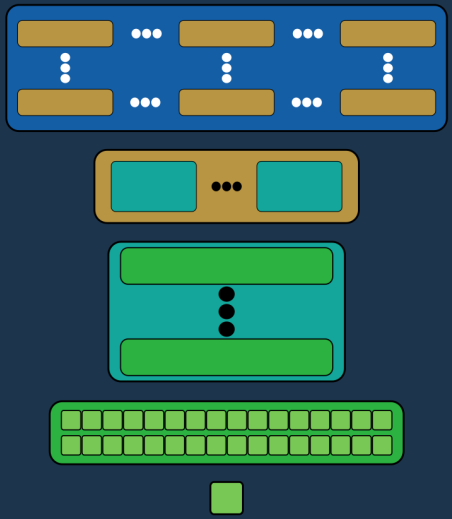
Cluster — Blocks Communication — Since : - CUDA **12** - **Hopper** — **Hopper** : 132 SMs **Ampere** : 108 SMs

Block — Streaming Multiprocessor — 2 - 4 **Warps** 64 - 128 **Cores** — **Asynchronous** Same Instruction Multiple Data

Warp — Hardware Scheduler — 32 **Threads** — **Synchronous** Same Instruction Multiple Data

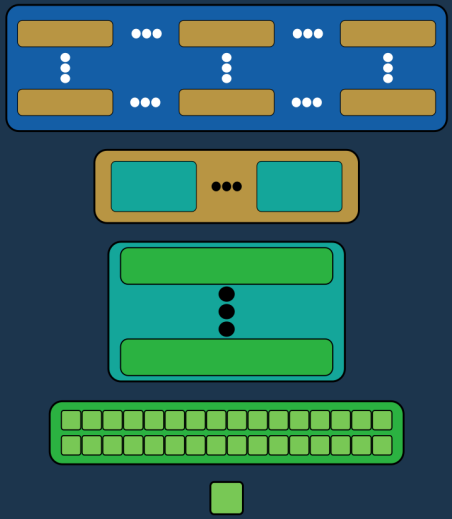
Thread — Core — Executes Computing Kernel

Threads Hierarchy



Grid	Full **GPU**		
Cluster	Blocks Communication	Since : - CUDA **12** - **Hopper**	**Hopper** : 132 SMs **Ampere** : 108 SMs
Block	Streaming Multiprocessor	2 - 4 **Warps** 64 - 128 **Cores**	**Asynchronous** Same Instruction Multiple Data
Warp	Hardware Scheduler	32 **Threads**	**Synchronous** Same Instruction Multiple Data
Thread	Core	Executes Computing Kernel	

Threads Hierarchy



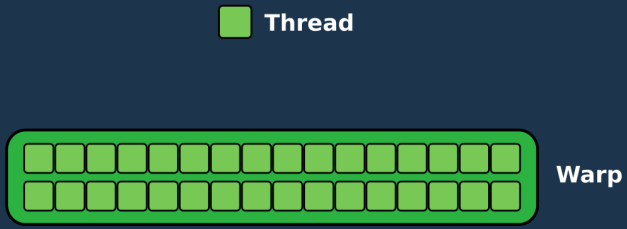
Grid	Full **GPU**		**Hopper** : 8448 Cores **Ampere** : 6912 Cores
Cluster	Blocks Communication	Since : - CUDA **12** - **Hopper**	**Hopper** : 132 SMs **Ampere** : 108 SMs
Block	Streaming Multiprocessor	2 - 4 **Warps** 64 - 128 **Cores**	**Asynchronous** Same Instruction Multiple Data
Warp	Hardware Scheduler	32 **Threads**	**Synchronous** Same Instruction Multiple Data
Thread	Core	Executes Computing Kernel	

Threads Execution

Threads Execution

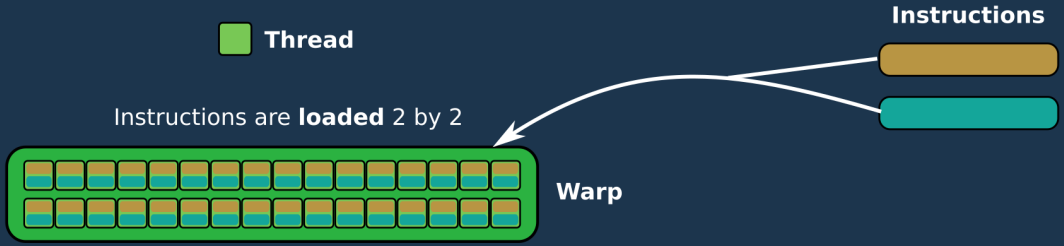
Thread

Threads Execution

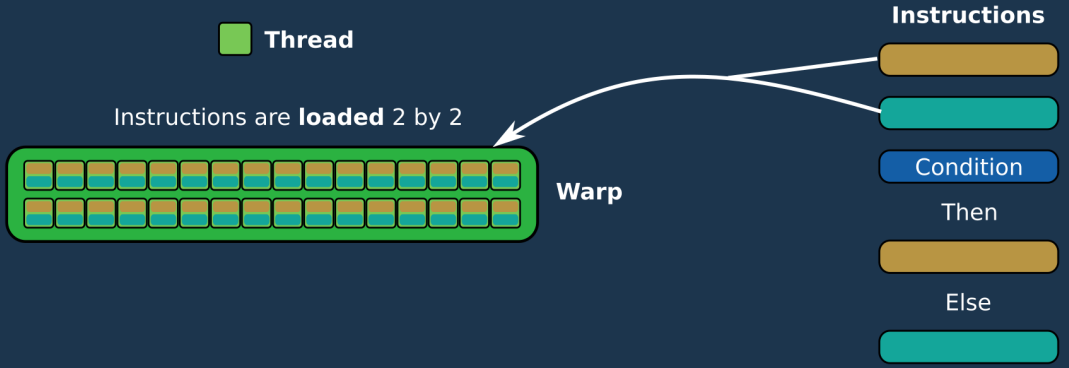


Threads Execution

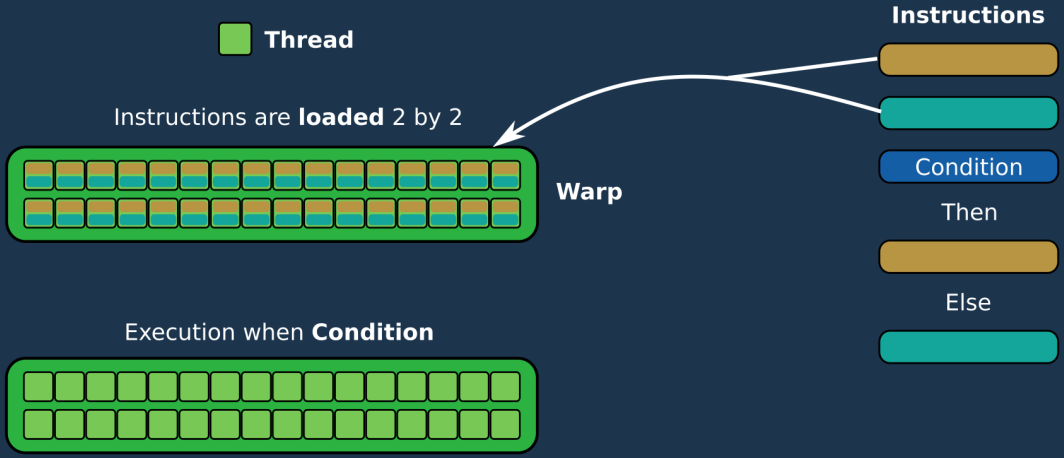
Threads Execution



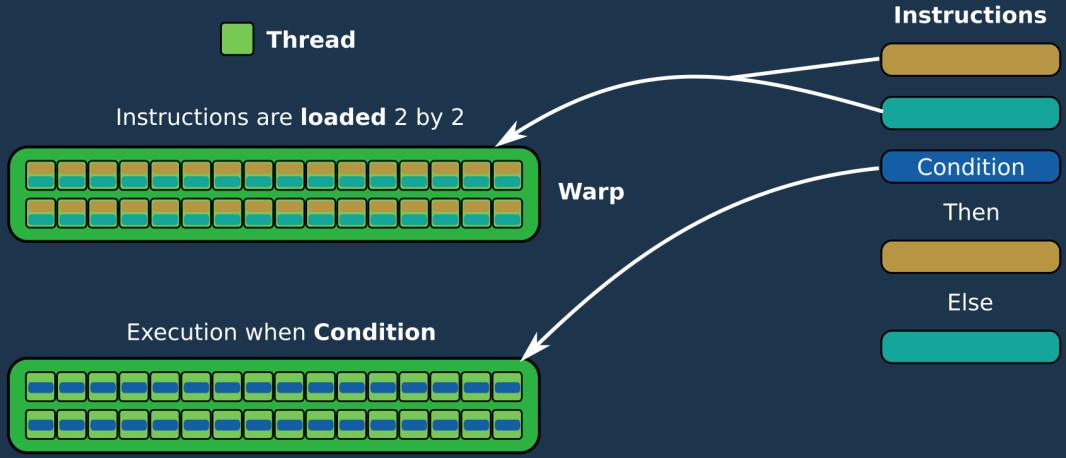
Threads Execution



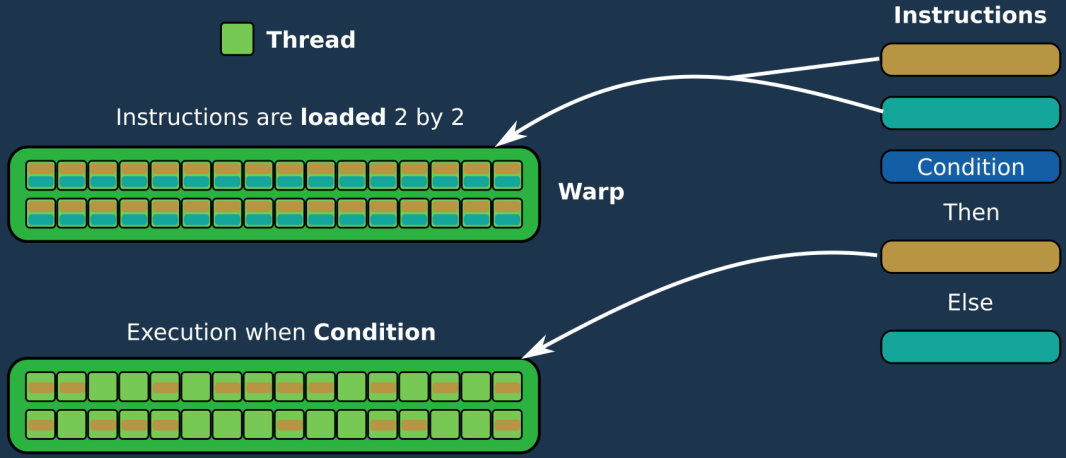
Threads Execution



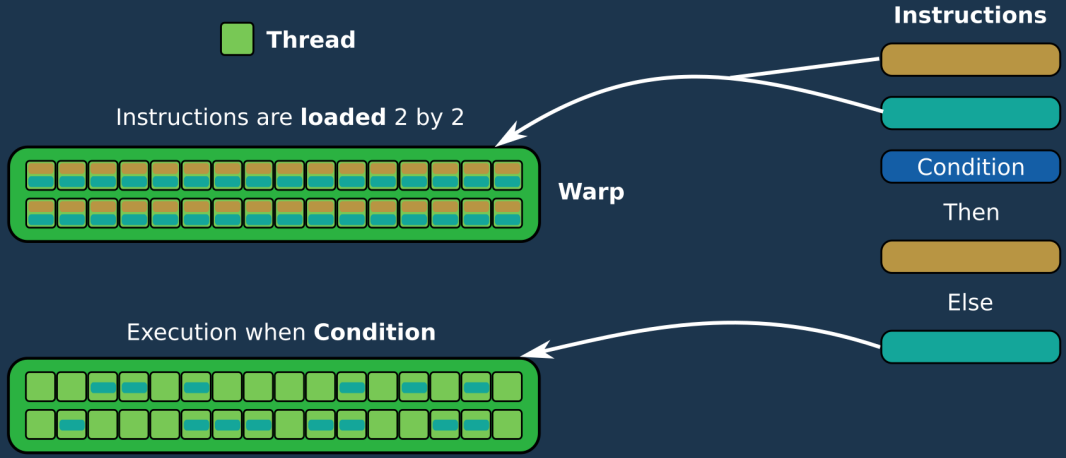
Threads Execution



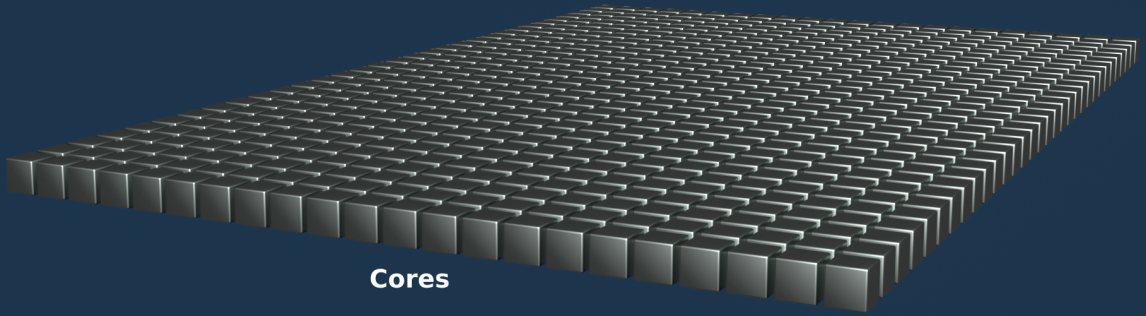
Threads Execution



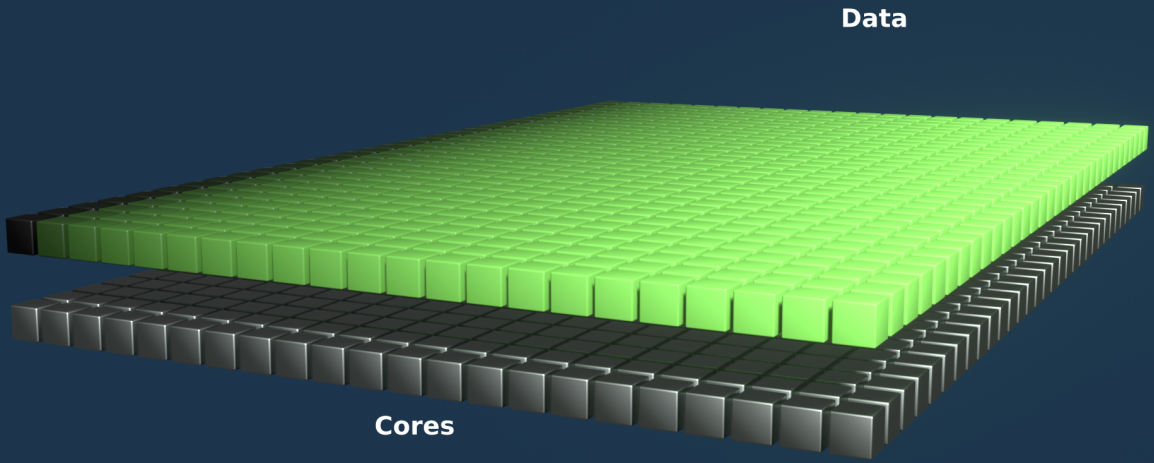
Threads Execution



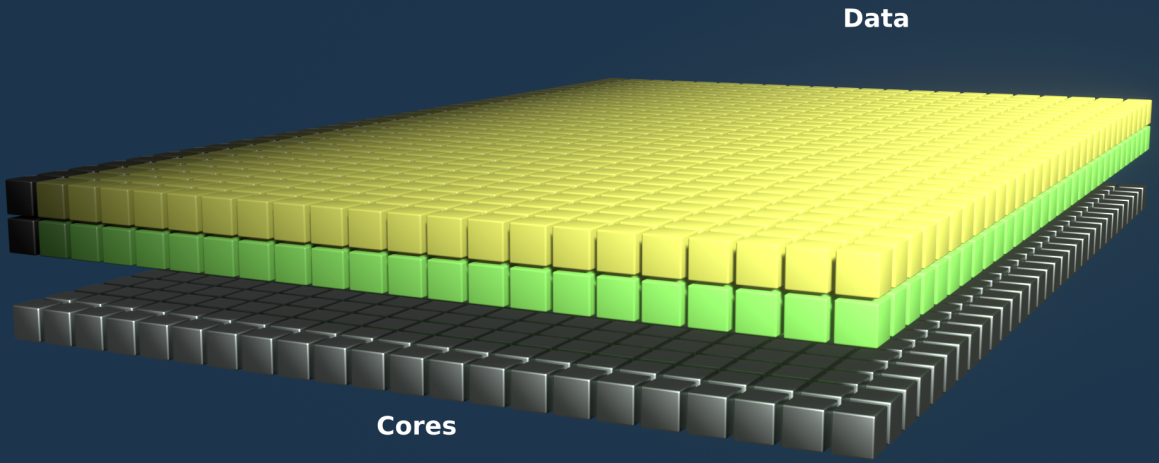
Thread Mapping



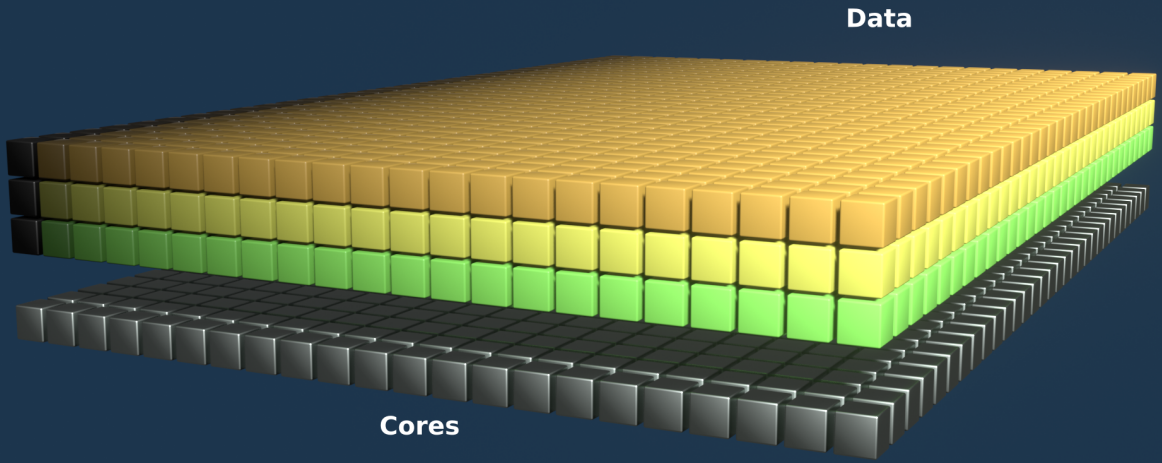
Thread Mapping



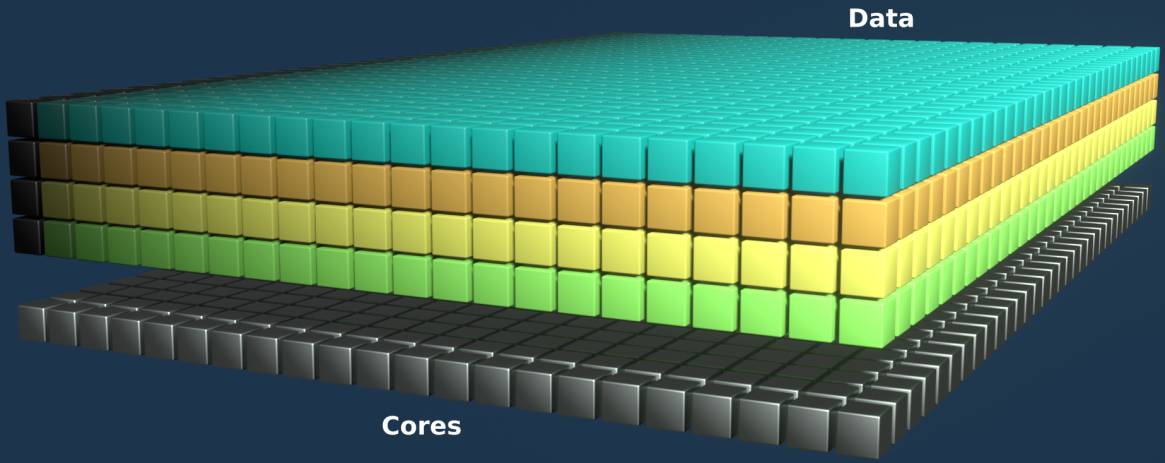
Thread Mapping



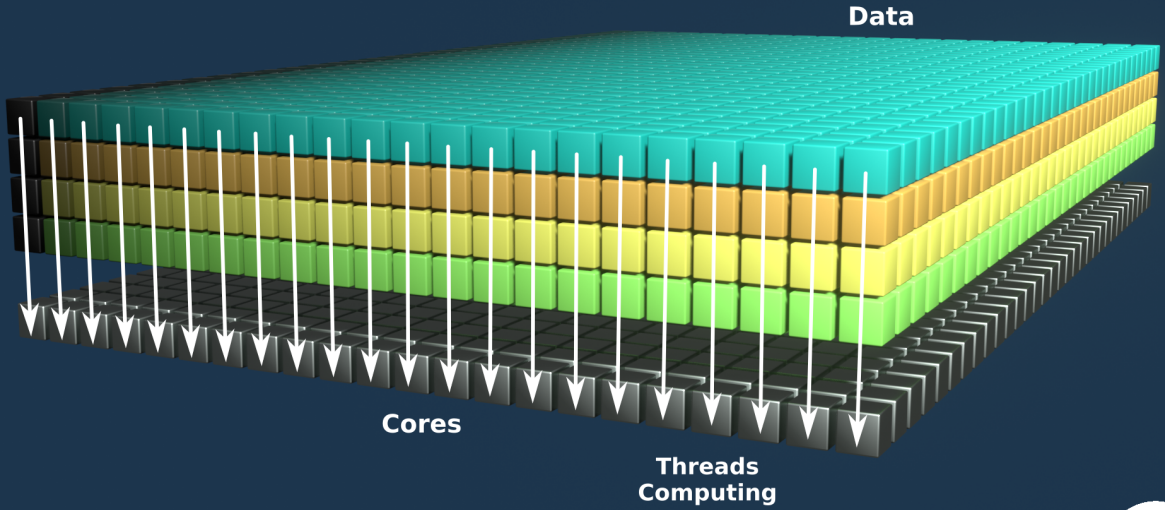
Thread Mapping



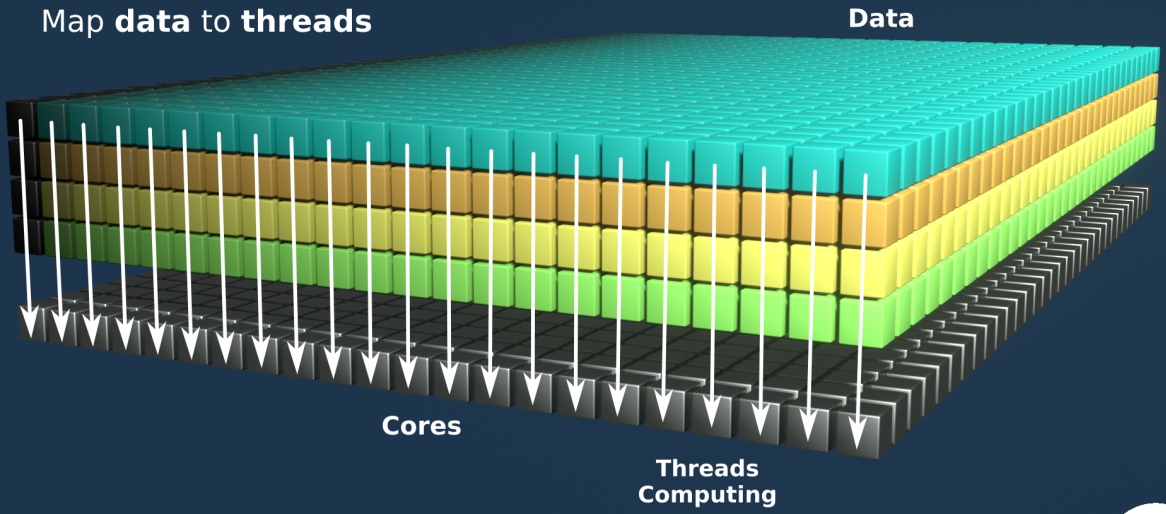
Thread Mapping



Thread Mapping



Thread Mapping



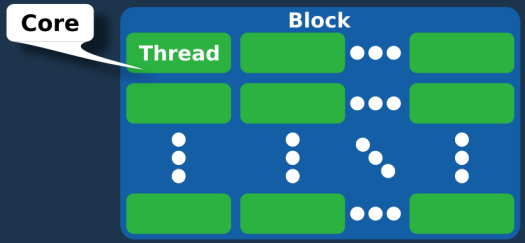
Data / Thread mapping

Data / Thread mapping

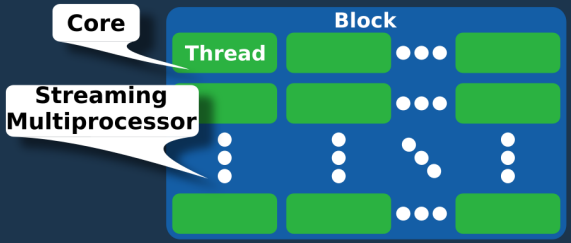
Thread

Data / Thread mapping

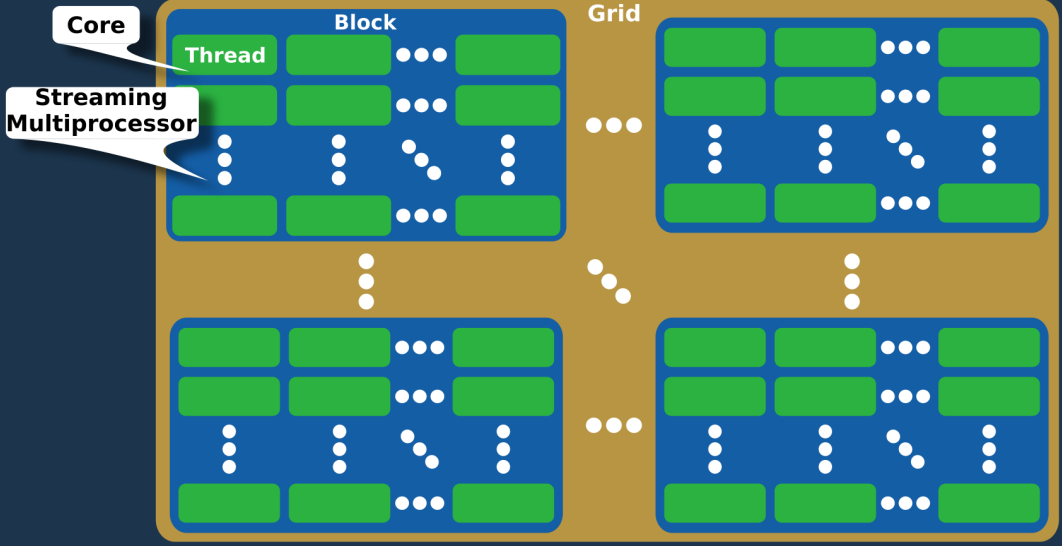
Data / Thread mapping



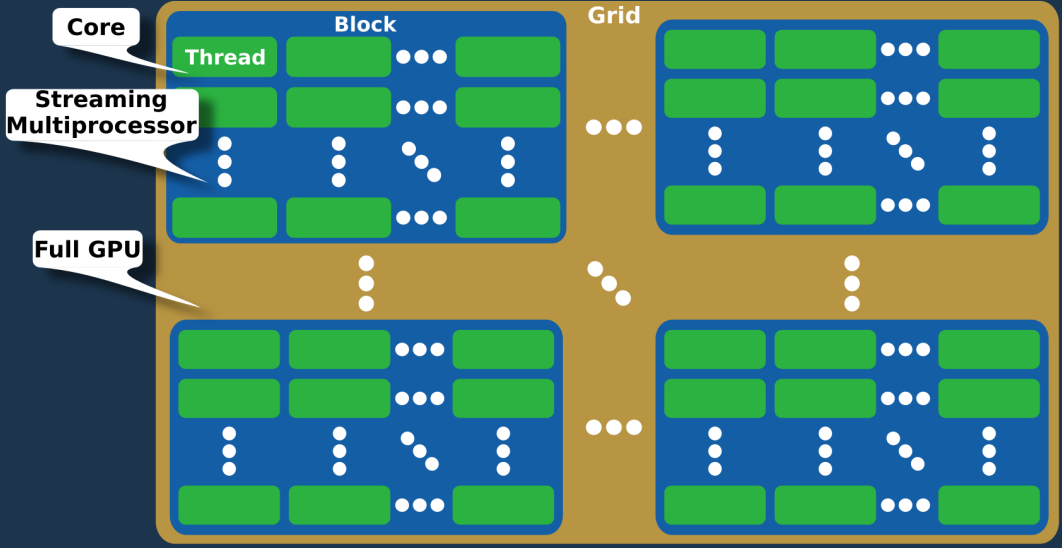
Data / Thread mapping



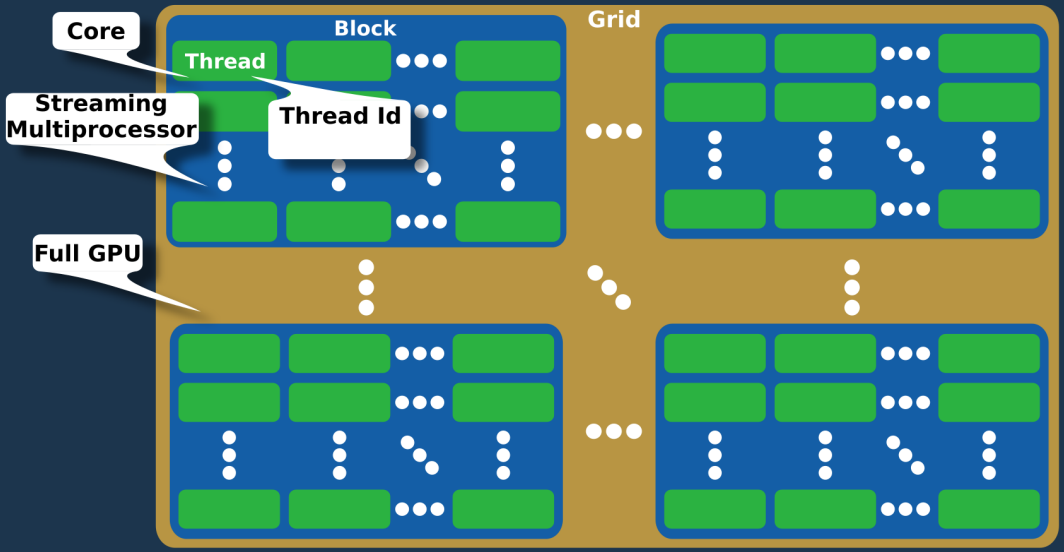
Data / Thread mapping



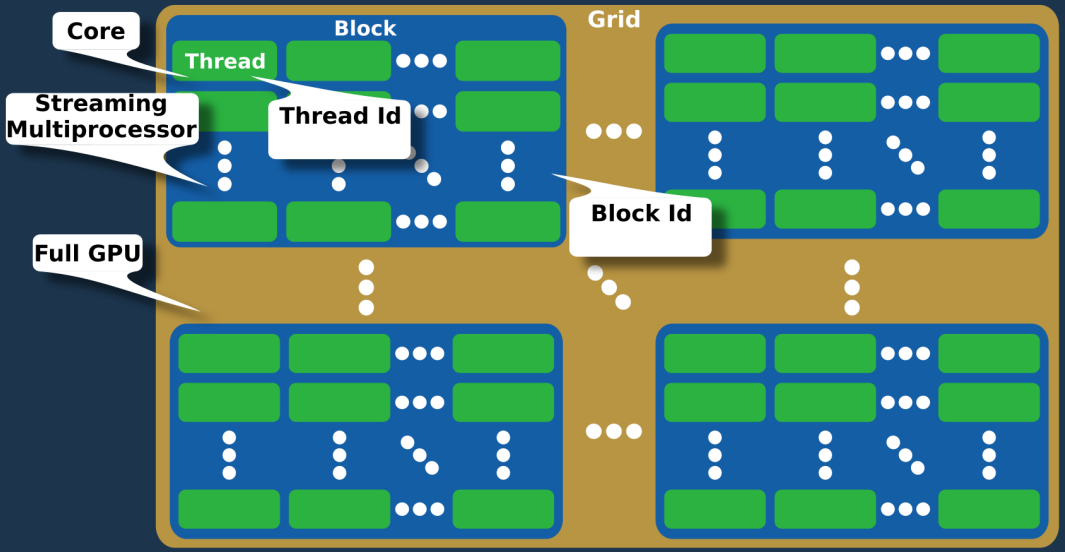
Data / Thread mapping



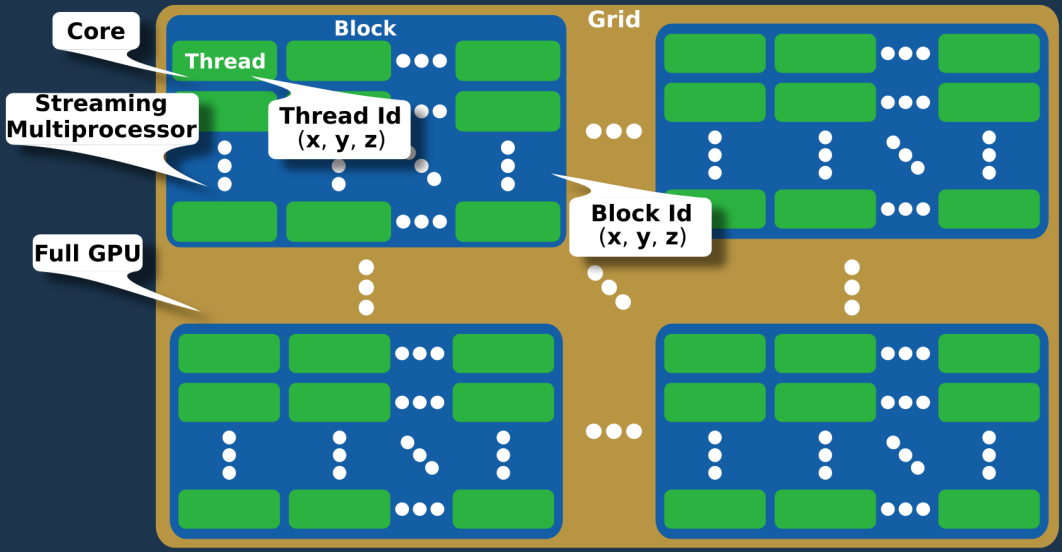
Data / Thread mapping



Data / Thread mapping



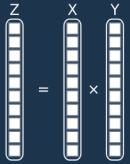
Data / Thread mapping



Block Size ?

Block Size ?

Hadamard product on **1000** elements



Block Size ?

Hadamard product on **1000** elements

Scenario #1

10 1d blocks of **size 100**

Block Size ?

Warp of **32 threads**

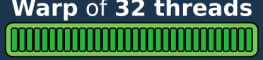
Hadamard product on **1000** elements

Scenario #1

10 1d blocks of **size 100**

Block Size ?

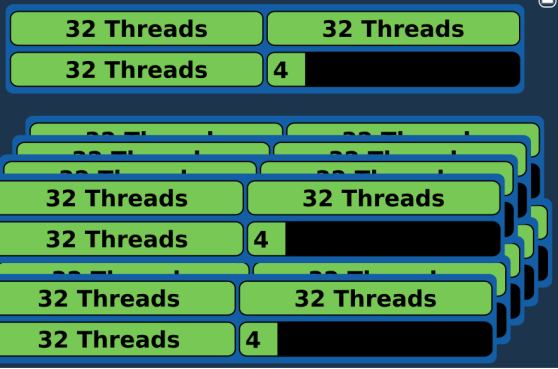
Warp of **32 threads**



Hadamard product on **1000** elements

Scenario #1

10 1d blocks of **size 100**



Block Size ?

Warp of **32 threads**

Hadamard product on **1000** elements

Use: 78%

Scenario #1

10 1d blocks of **size 100**

Block Size ?

Warp of **32 threads**

Hadamard product on **1000** elements

Use: 78%

Scenario #1

10 1d blocks of **size 100**

Scenario #2

8 1d blocks of **size 128**

Block Size ?

Warp of **32 threads**

Hadamard product on **1000** elements

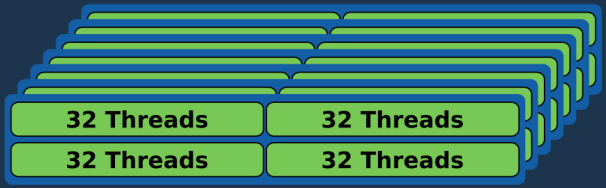
Use: 78%

Scenario #1

10 1d blocks of **size 100**

Scenario #2

8 1d blocks of **size 128**



Block Size ?

Warp of **32 threads**

Hadamard product on **1000** elements

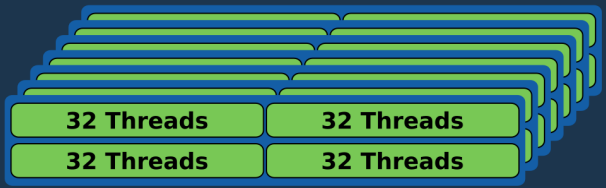
Use: 78%

Scenario #1

10 1d blocks of **size 100**

Scenario #2

8 1d blocks of **size 128**



Block Size ?

Warp of **32 threads**

Hadamard product on **1000** elements

Scenario #1

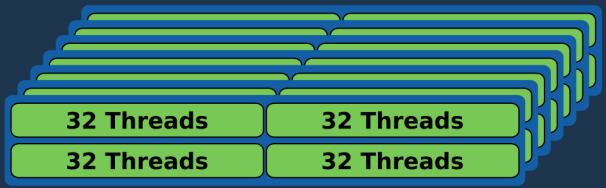
Use: 78%

10 1d blocks of **size 100**

Scenario #2

Use: 98%

8 1d blocks of **size 128**



Warp Read / Write

Warp Read / Write

1 sector : 32 bytes

Cache line size :
- 128 bytes
- 4 sectors

Warp Read / Write

1 sector : 32 bytes

Cache line size :
- 128 bytes
- 4 sectors

4-bytes elements access -> 4 sectors

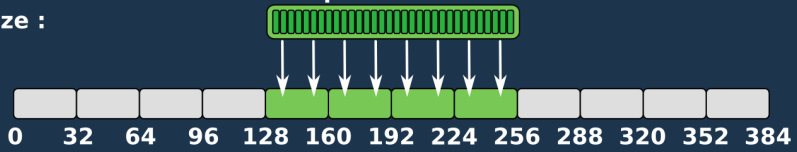
Warp of **32 threads**

Warp Read / Write

1 sector : 32 bytes

Cache line size :
- 128 bytes
- 4 sectors

4-bytes elements access -> 4 sectors

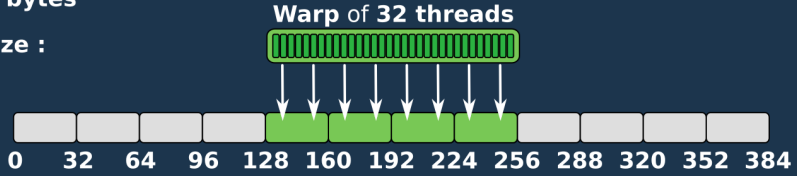


Warp Read / Write

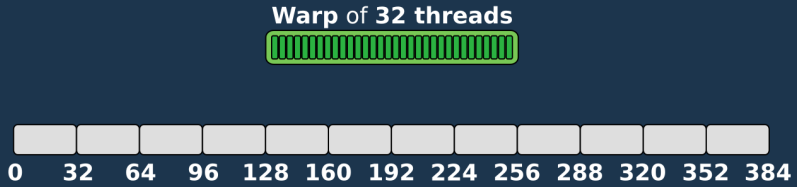
1 sector : 32 bytes

Cache line size :
- 128 bytes
- 4 sectors

4-bytes elements access -> 4 sectors



8-bytes elements access -> 8 sectors

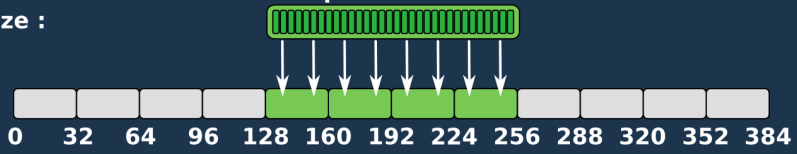


Warp Read / Write

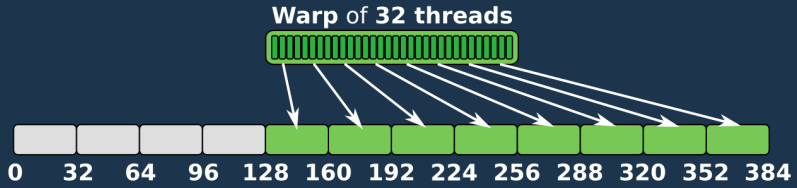
1 sector : 32 bytes

Cache line size :
- 128 bytes
- 4 sectors

4-bytes elements access -> 4 sectors



8-bytes elements access -> 8 sectors

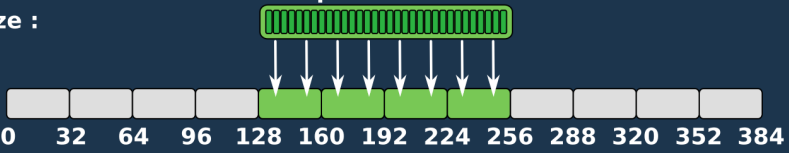


Warp Read / Write

1 sector : 32 bytes

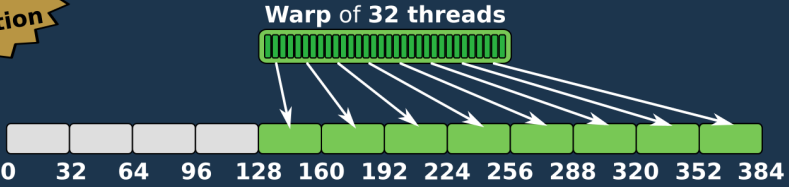
Cache line size :
- 128 bytes
- 4 sectors

4-bytes elements access -> 4 sectors



Really close to CPU Vectorization

8-bytes elements access -> 8 sectors



Memory Layout

Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Array of Structure (AoS)

```
struct Coeff{
    float u, v, w;
    float x[8], y[8], z;
};
```

80 Bytes

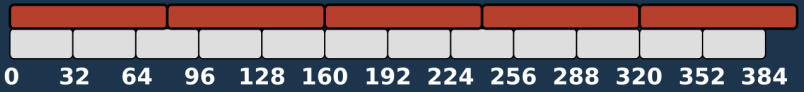
Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Array of Structure (AoS)
Warp of **32 threads**

```
struct Coeff{
    float u, v, w;
    float x[8], y[8], z;
};
```

80 Bytes



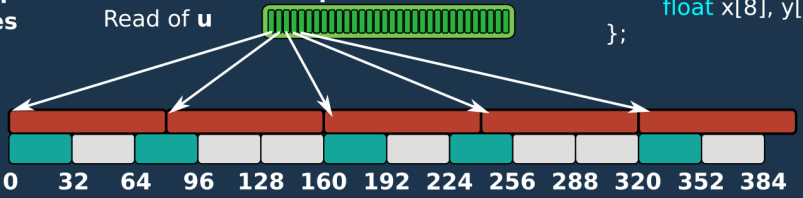
Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Array of Structure (AoS)

Warp of **32 threads**

Read of **u**




```
struct Coeff{
    float u, v, w;
    float x[8], y[8], z;
};
```

80 Bytes

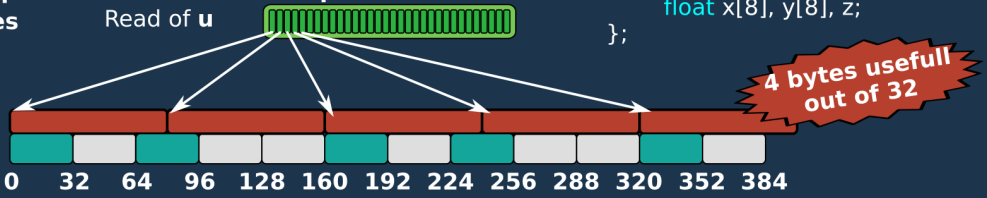
Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Array of Structure (AoS)

Warp of **32 threads**

Read of **u**




```
struct Coeff{
    float u, v, w;
    float x[8], y[8], z;
};
```

80 Bytes

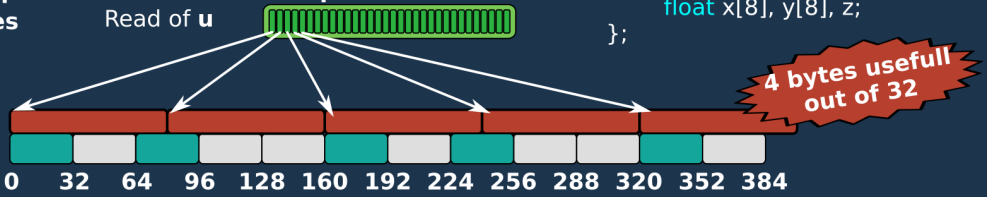
Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Array of Structure (AoS)

Warp of **32 threads**

Read of **u**




```
struct Coeff{
        float u, v, w;
        float x[8], y[8], z;
};
```

80 Bytes

Strucutre of Array (SoA)

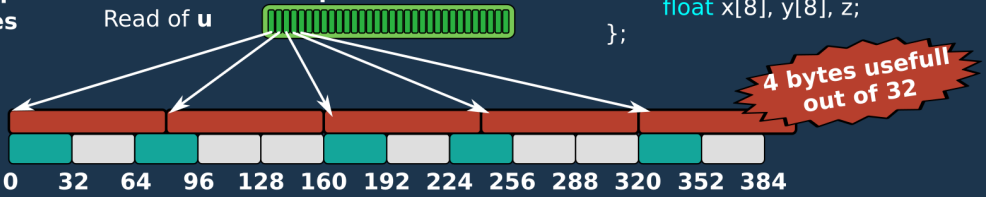
```
struct Coeff{
        float *u, *v, *w;
        float *x0, ... *x7, *y0, ... *y7, *z;
};
```

Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Array of Structure (AoS)
Warp of **32 threads**

Read of **u**



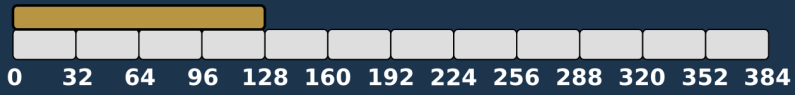

```
struct Coeff{
    float u, v, w;
    float x[8], y[8], z;
};
```

80 Bytes

4 bytes usefull out of 32

Strucutre of Array (SoA)
Warp of **32 threads**

```
struct Coeff{
    float *u, *v, *w;
    float *x0, ... *x7, *y0, ... *y7, *z;
};
```

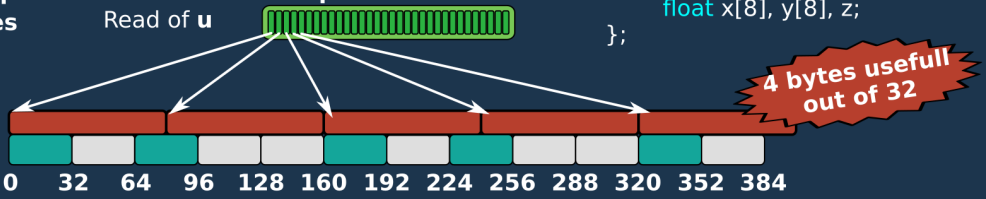


Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Array of Structure (AoS)
Warp of **32 threads**

Read of **u**

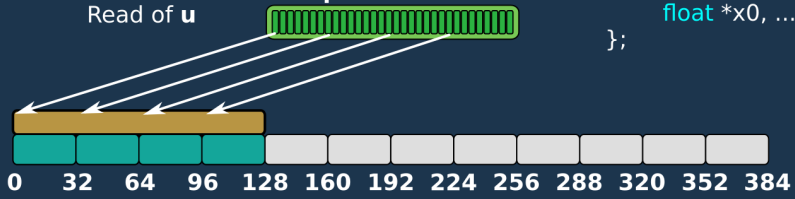



```
struct Coeff{
    float u, v, w;
    float x[8], y[8], z;
};
```

80 Bytes

Strucutre of Array (SoA)
Warp of **32 threads**

Read of **u**



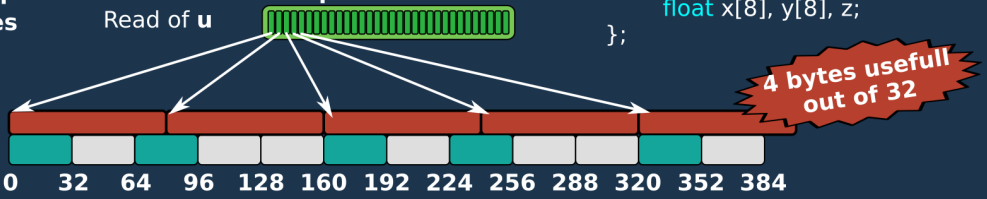

```
struct Coeff{
    float *u, *v, *w;
    float *x0, ... *x7, *y0, ... *y7, *z;
};
```

Memory Layout

Read / Write :
- 1 sector
- 32 bytes

Array of Structure (AoS)
Warp of **32 threads**

Read of **u**

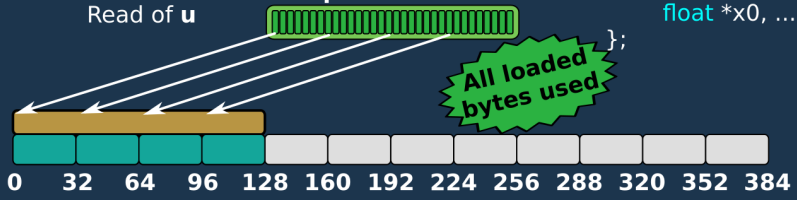


struct Coeff{
    float u, v, w;
    float x[8], y[8], z;
};

80 Bytes

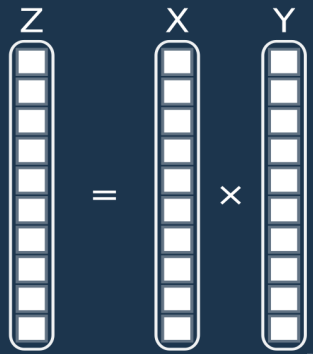
Strucutre of Array (SoA)
Warp of **32 threads**

Read of **u**

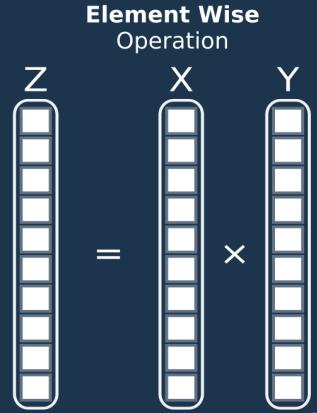


struct Coeff{
    float *u, *v, *w;
    float *x0, ... *x7, *y0, ... *y7, *z;
};

Hadamard Product CUDA

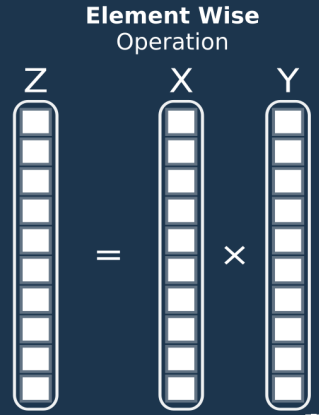


Hadamard Product CUDA



Hadamard Product CUDA

```
__global__ void hadamard_product_kernel(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement){
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        vecResult[i] = vecLeft[i] * vecRight[i];
}
```

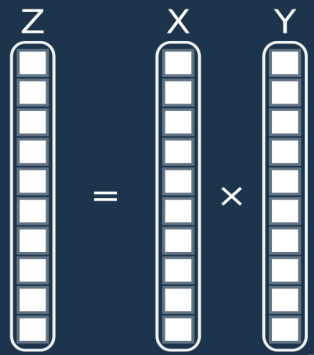


Hadamard Product CUDA

```cpp
__global__ void hadamard_product_kernel(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    vecResult[i] = vecLeft[i] * vecRight[i];
}
```

Execution on
Device or **Host**

Element Wise
Operation



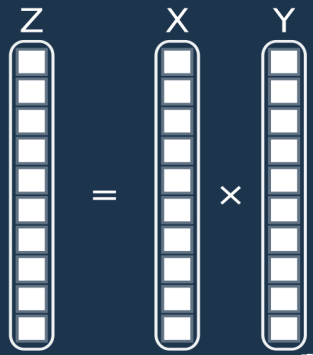
Hadamard Product CUDA

```
__global__ void hadamard_product_kernel(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    vecResult[i] = vecLeft[i] * vecRight[i];
}
```

Execution on **Device** or **Host**

Thread position

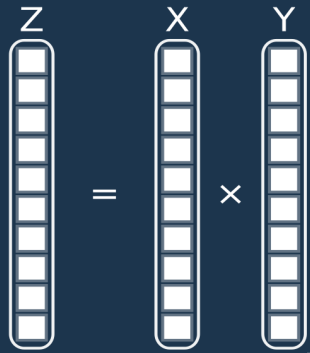
Element Wise Operation



Hadamard Product CUDA

```
void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                           int maxNbThreadPerBlockX)
{
        //Allocations
        float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
        int sizeByte = nbElement*sizeof(float);
        cudaMalloc((void**)&vecResultd, sizeByte);
        cudaMalloc((void**)&vecLeftd, sizeByte);
        cudaMalloc((void**)&vecRightd, sizeByte);

        //Block and grid size
        int dimGridX = 1;
        int dimBlockX = 1;
        if(nbElement > maxNbThreadPerBlockX){
                dimBlockX = maxNbThreadPerBlockX;
                dimGridX = nbElement/dimBlockX;
        }else{
                dimBlockX = nbElement;
        }

        //Transfer to Device
        cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE
        cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE

        dim3 dimGrid(dimGridX, 1, 1);
        dim3 dimBlock(dimBlockX, 1, 1);
        hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
        PLIB_CUDA_CHECK_FILE
        //Transfer to Host
        cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
        PLIB_CUDA_CHECK_FILE
        //Free memory
        cudaFree(vecRightd);
        cudaFree(vecLeftd);
        cudaFree(vecResultd);
}
```

Element Wise
Operation

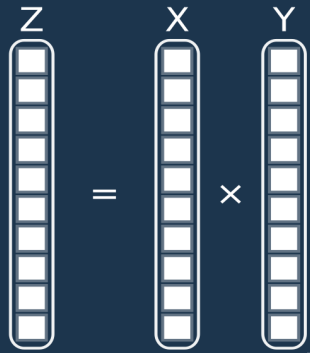


Hadamard Product CUDA

```cpp
void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                                        int maxNbThreadPerBlockX)
{
        //Allocations
        float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
        int sizeByte = nbElement*sizeof(float);
        cudaMalloc((void**)&vecResultd, sizeByte);
        cudaMalloc((void**)&vecLeftd, sizeByte);
        cudaMalloc((void**)&vecRightd, sizeByte);

        //Block and grid size
        int dimGridX = 1;
        int dimBlockX = 1;
        if(nbElement > maxNbThreadPerBlockX){
                dimBlockX = maxNbThreadPerBlockX;
                dimGridX = nbElement/dimBlockX;
        }else{
                dimBlockX = nbElement;
        }

        //Transfer to Device
        cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE
        cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE

        dim3 dimGrid(dimGridX, 1, 1);
        dim3 dimBlock(dimBlockX, 1, 1);
        hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
        PLIB_CUDA_CHECK_FILE
        //Transfer to Host
        cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
        PLIB_CUDA_CHECK_FILE
        //Free memory
        cudaFree(vecRightd);
        cudaFree(vecLeftd);
        cudaFree(vecResultd);
}
```

Allocation on **GPU**

Element Wise Operation



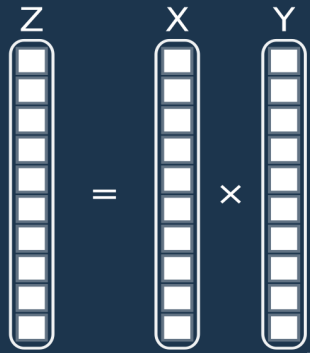
Hadamard Product CUDA

```cpp
void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                                        int maxNbThreadPerBlockX)
{
        //Allocations
        float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
        int sizeByte = nbElement*sizeof(float);
        cudaMalloc((void**)&vecResultd, sizeByte);
        cudaMalloc((void**)&vecLeftd, sizeByte);
        cudaMalloc((void**)&vecRightd, sizeByte);

        //Block and grid size
        int dimGridX = 1;
        int dimBlockX = 1;
        if(nbElement > maxNbThreadPerBlockX){
                dimBlockX = maxNbThreadPerBlockX;
                dimGridX = nbElement/dimBlockX;
        }else{
                dimBlockX = nbElement;
        }

        //Transfer to Device
        cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE
        cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE

        dim3 dimGrid(dimGridX, 1, 1);
        dim3 dimBlock(dimBlockX, 1, 1);
        hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
        PLIB_CUDA_CHECK_FILE
        //Transfer to Host
        cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
        PLIB_CUDA_CHECK_FILE
        //Free memory
        cudaFree(vecRightd);
        cudaFree(vecLeftd);
        cudaFree(vecResultd);
}
```

Allocation on **GPU**

Free on **GPU**

Element Wise Operation



Hadamard Product CUDA

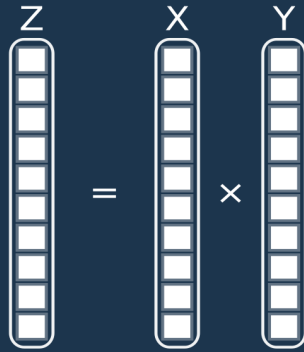
```
void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                           int maxNbThreadPerBlockX)
{
        //Allocations
        float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
        int sizeByte = nbElement*sizeof(float);
        cudaMalloc((void**)&vecResultd, sizeByte);
        cudaMalloc((void**)&vecLeftd, sizeByte);
        cudaMalloc((void**)&vecRightd, sizeByte);

        //Block and grid size
        int dimGridX = 1;
        int dimBlockX = 1;
        if(nbElement > maxNbThreadPerBlockX){
                dimBlockX = maxNbThreadPerBlockX;
                dimGridX = nbElement/dimBlockX;
        }else{
                dimBlockX = nbElement;
        }

        //Transfer to Device
        cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE
        cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE

        dim3 dimGrid(dimGridX, 1, 1);
        dim3 dimBlock(dimBlockX, 1, 1);
        hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
        PLIB_CUDA_CHECK_FILE
        //Transfer to Host
        cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
        PLIB_CUDA_CHECK_FILE
        //Free memory
        cudaFree(vecRightd);
        cudaFree(vecLeftd);
        cudaFree(vecResultd);
}
```

Out of this function **if many calls**

Allocation on **GPU**

Free on **GPU**

Element Wise Operation



Hadamard Product CUDA

```
void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                           int maxNbThreadPerBlockX)
{
        //Allocations
        float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
        int sizeByte = nbElement*sizeof(float);
        cudaMalloc((void**)&vecResultd, sizeByte);
        cudaMalloc((void**)&vecLeftd, sizeByte);
        cudaMalloc((void**)&vecRightd, sizeByte);

        //Block and grid size
        int dimGridX = 1;
        int dimBlockX = 1;
        if(nbElement > maxNbThreadPerBlockX){
                dimBlockX = maxNbThreadPerBlockX;
                dimGridX = nbElement/dimBlockX;
        }else{
                dimBlockX = nbElement;
        }

        //Transfer to Device
        cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE
        cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE

        dim3 dimGrid(dimGridX, 1, 1);
        dim3 dimBlock(dimBlockX, 1, 1);
        hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
        PLIB_CUDA_CHECK_FILE
        //Transfer to Host
        cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
        PLIB_CUDA_CHECK_FILE
        //Free memory
        cudaFree(vecRightd);
        cudaFree(vecLeftd);
        cudaFree(vecResultd);
}
```

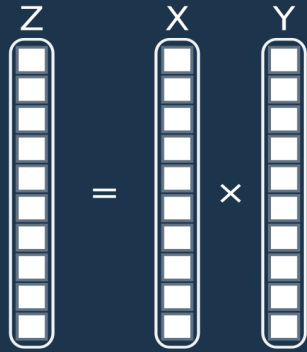
Out of this function **if many calls**

Allocation on **GPU**

Transfer data to **GPU**

Free on **GPU**

Element Wise Operation



Hadamard Product CUDA

```
void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                           int maxNbThreadPerBlockX)
{
        //Allocations
        float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
        int sizeByte = nbElement*sizeof(float);
        cudaMalloc((void**)&vecResultd, sizeByte);
        cudaMalloc((void**)&vecLeftd, sizeByte);
        cudaMalloc((void**)&vecRightd, sizeByte);

        //Block and grid size
        int dimGridX = 1;
        int dimBlockX = 1;
        if(nbElement > maxNbThreadPerBlockX){
                dimBlockX = maxNbThreadPerBlockX;
                dimGridX = nbElement/dimBlockX;
        }else{
                dimBlockX = nbElement;
        }

        //Transfer to Device
        cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE
        cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE

        dim3 dimGrid(dimGridX, 1, 1);
        dim3 dimBlock(dimBlockX, 1, 1);
        hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
        PLIB_CUDA_CHECK_FILE
        //Transfer to Host
        cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
        PLIB_CUDA_CHECK_FILE
        //Free memory
        cudaFree(vecRightd);
        cudaFree(vecLeftd);
        cudaFree(vecResultd);
}
```

Out of this function **if many calls**

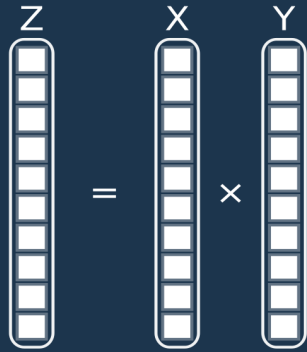
Allocation on **GPU**

Transfer data to **GPU**

Transfer result back to **host**

Free on **GPU**

Element Wise Operation



Hadamard Product CUDA

```
void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                           int maxNbThreadPerBlockX)
{
    //Allocations
    float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
    int sizeByte = nbElement*sizeof(float);
    cudaMalloc((void**)&vecResultd, sizeByte);
    cudaMalloc((void**)&vecLeftd, sizeByte);
    cudaMalloc((void**)&vecRightd, sizeByte);

    //Block and grid size
    int dimGridX = 1;
    int dimBlockX = 1;
    if(nbElement > maxNbThreadPerBlockX){
        dimBlockX = maxNbThreadPerBlockX;
        dimGridX = nbElement/dimBlockX;
    }else{
        dimBlockX = nbElement;
    }

    //Transfer to Device
    cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
    PLIB_CUDA_CHECK_FILE
    cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
    PLIB_CUDA_CHECK_FILE

    dim3 dimGrid(dimGridX, 1, 1);
    dim3 dimBlock(dimBlockX, 1, 1);
    hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
    PLIB_CUDA_CHECK_FILE
    //Transfer to Host
    cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
    PLIB_CUDA_CHECK_FILE
    //Free memory
    cudaFree(vecRightd);
    cudaFree(vecLeftd);
    cudaFree(vecResultd);
}
```

Out of this function **if many calls**

Allocation on **GPU**

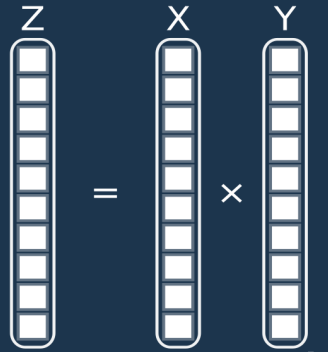
Determine Grid and **Block** size

Transfer data to **GPU**

Transfer result back to **host**

Free on **GPU**

Element Wise Operation



Hadamard Product CUDA

```
void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                           int maxNbThreadPerBlockX)
{
    //Allocations
    float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
    int sizeByte = nbElement*sizeof(float);
    cudaMalloc((void**)&vecResultd, sizeByte);
    cudaMalloc((void**)&vecLeftd, sizeByte);
    cudaMalloc((void**)&vecRightd, sizeByte);

    //Block and grid size
    int dimGridX = 1;
    int dimBlockX = 1;
    if(nbElement > maxNbThreadPerBlockX){
        dimBlockX = maxNbThreadPerBlockX;
        dimGridX = nbElement/dimBlockX;
    }else{
        dimBlockX = nbElement;
    }

    //Transfer to Device
    cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
    PLIB_CUDA_CHECK_FILE
    cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
    PLIB_CUDA_CHECK_FILE

    dim3 dimGrid(dimGridX, 1, 1);
    dim3 dimBlock(dimBlockX, 1, 1);
    hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
    PLIB_CUDA_CHECK_FILE
    //Transfer to Host
    cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
    PLIB_CUDA_CHECK_FILE
    //Free memory
    cudaFree(vecRightd);
    cudaFree(vecLeftd);
    cudaFree(vecResultd);
}
```

Out of this function **if many calls**

Allocation on **GPU**

Determine Grid and **Block** size

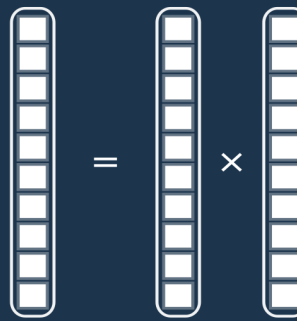
Transfer data to **GPU**

Call Kernel

Transfer result back to **host**

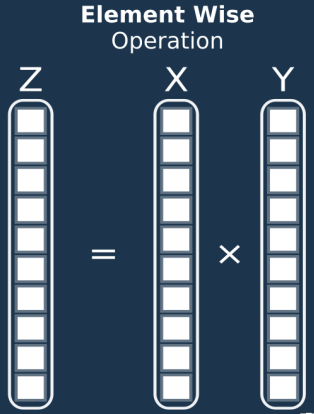
Free on **GPU**

Element Wise Operation



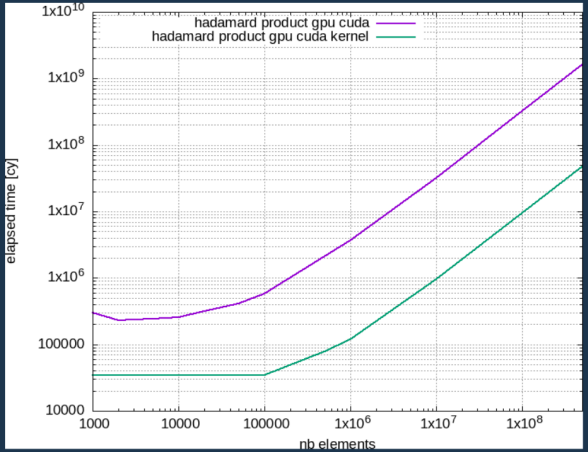
Hadamard Product CUDA

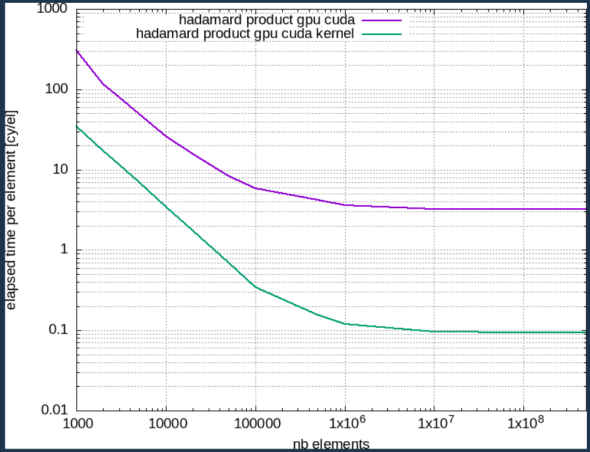
```cpp
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void hadamard_product_kernel(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement){
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        vecResult[i] = vecLeft[i] * vecRight[i];
}

void hadamard_product_cuda(float * vecResult, const float * vecLeft, const float * vecRight, int nbElement,
                                        int maxNbThreadPerBlockX)
{
        //Allocations
        float *vecResultd = NULL, *vecLeftd = NULL, *vecRightd = NULL;
        int sizeByte = nbElement*sizeof(float);
        cudaMalloc((void**)&vecResultd, sizeByte);
        cudaMalloc((void**)&vecLeftd, sizeByte);
        cudaMalloc((void**)&vecRightd, sizeByte);

        //Block and grid size
        int dimGridX = 1;
        int dimBlockX = 1;
        if(nbElement > maxNbThreadPerBlockX){
                dimBlockX = maxNbThreadPerBlockX;
                dimGridX = nbElement/dimBlockX;
        }else{
                dimBlockX = nbElement;
        }

        //Transfer to Device
        cudaMemcpy(vecLeftd, vecLeft, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE
        cudaMemcpy(vecRightd, vecRight, sizeByte, cudaMemcpyHostToDevice);
        PLIB_CUDA_CHECK_FILE

        dim3 dimGrid(dimGridX, 1, 1);
        dim3 dimBlock(dimBlockX, 1, 1);
        hadamard_product_kernel<<<dimGrid, dimBlock>>>(vecResultd, vecLeftd, vecRightd, nbElement);
        PLIB_CUDA_CHECK_FILE
        //Transfer to Host
        cudaMemcpy(vecResult, vecResultd, sizeByte, cudaMemcpyDeviceToHost);
        PLIB_CUDA_CHECK_FILE
        //Free memory
        cudaFree(vecRightd);
        cudaFree(vecLeftd);
        cudaFree(vecResultd);
}
```



Perf : Hadamard Product CUDA

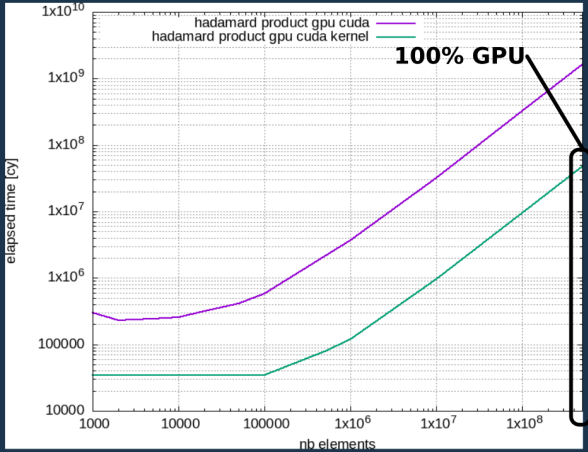
On A3000 GPU

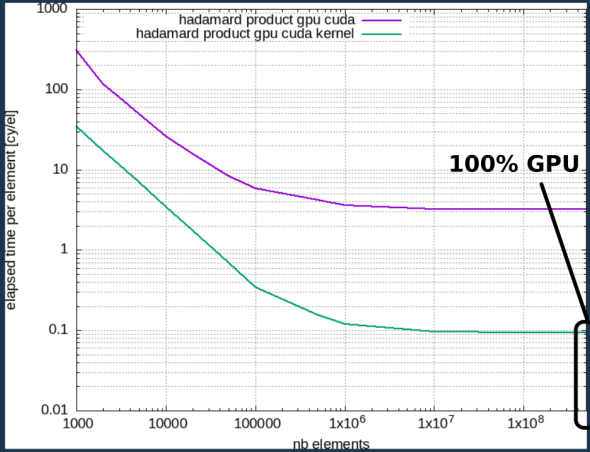




Perf : Hadamard Product CUDA

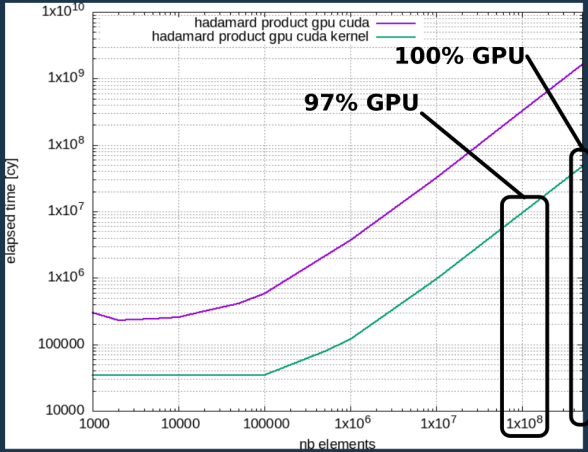
On A3000 GPU

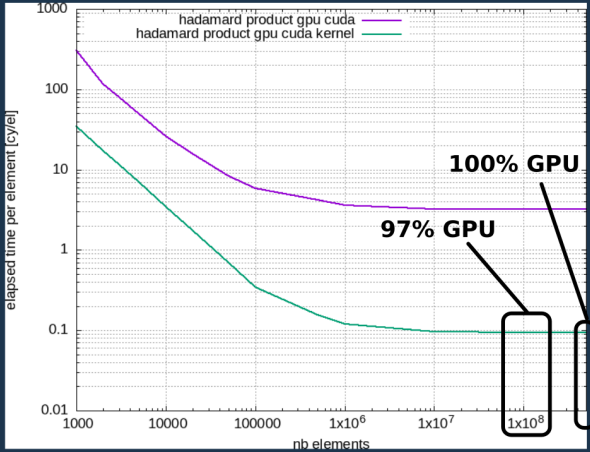




Perf : Hadamard Product CUDA

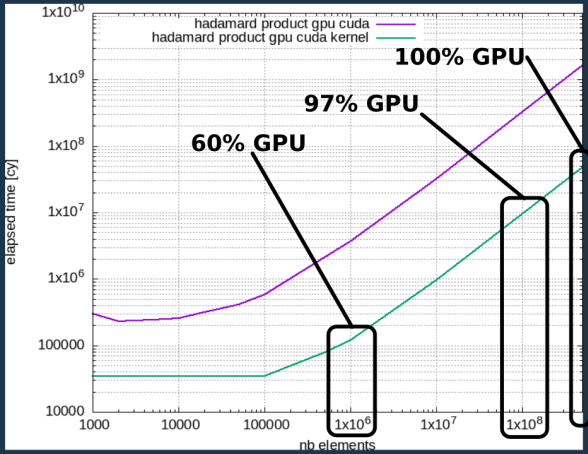
On A3000 GPU

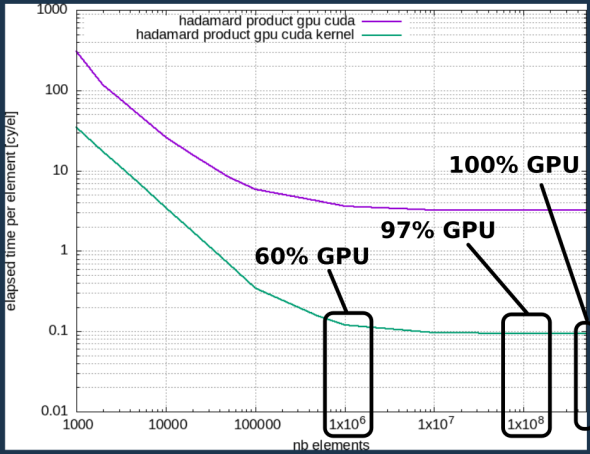




Perf : Hadamard Product CUDA

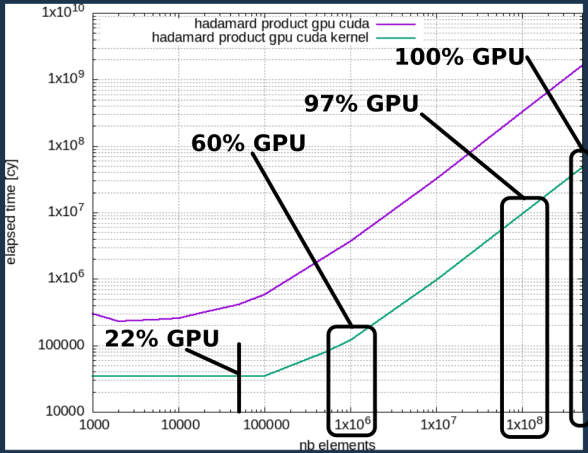
On A3000 GPU

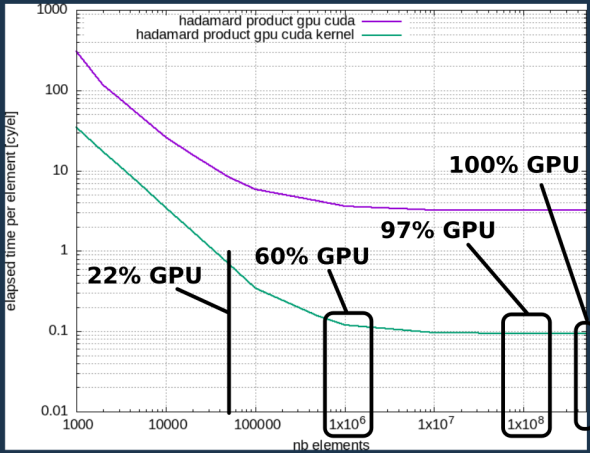




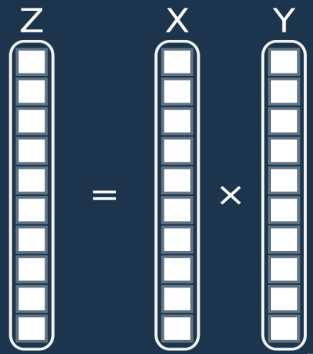
Perf : Hadamard Product CUDA

On A3000 GPU

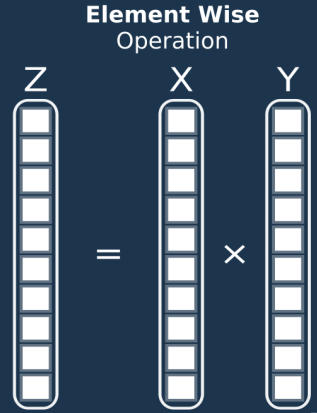




Hadamard Product Thrust



Hadamard Product Thrust



Hadamard Product Thrust

```
int main(int argc, char** argv){
        size_t nbElement(500000000);

        thrust::host_vector<float> h_vecX(nbElement), h_vecY(nbElement);
        for(size_t i(0lu); i < nbElement; ++i){
                h_vecX[i] = i*19lu%11;
                h_vecY[i] = i*27lu%19;
        }
        // Transfer data to the device.
        thrust::device_vector<float> d_vecX = h_vecX, d_vecY = h_vecY, d_vecRes(nbElement);

        thrust::transform(d_vecX.begin(), d_vecX.end(), d_vecY.begin(), d_vecRes.begin(),
                [=] __host__ __device__ (const float& x, const float& y){
                        return x * y;
                });

        // Transfer data to the host
        thrust::host_vector<float> h_vecRes = d_vecRes;

        std::cout << "x = " << h_vecX.front() << ", y = " << h_vecY.front() << ", res = " << h_vecRes.front() << std::endl;
        return 0;
}
```

Hadamard Product Thrust

```cpp
int main(int argc, char** argv){
        size_t nbElement(500000000);

        thrust::host_vector<float> h_vecX(nbElement), h_vecY(nbElement);
        for(size_t i(0lu); i < nbElement; ++i){
                h_vecX[i] = i*19lu%11;
                h_vecY[i] = i*27lu%19;
        }
        // Transfer data to the device.
        thrust::device_vector<float> d_vecX = h_vecX, d_vecY = h_vecY, d_vecRes(nbElement);

        thrust::transform(d_vecX.begin(), d_vecX.end(), d_vecY.begin(), d_vecRes.begin(),
                [=] __host__ __device__ (const float& x, const float& y){
                        return x * y;
                });

        // Transfer data to the host
        thrust::host_vector<float> h_vecRes = d_vecRes;

        std::cout << "x = " << h_vecX.front() << ", y = " << h_vecY.front() << ", res = " << h_vecRes.front() << std::endl;
        return 0;
}
```

Initialisation

Hadamard Product Thrust

```cpp
int main(int argc, char** argv){
        size_t nbElement(500000000);

        thrust::host_vector<float> h_vecX(nbElement), h_vecY(nbElement);
        for(size_t i(0lu); i < nbElement; ++i){
                h_vecX[i] = i*19lu%11;
                h_vecY[i] = i*27lu%19;
        }
        // Transfer data to the device.
        thrust::device_vector<float> d_vecX = h_vecX, d_vecY = h_vecY, d_vecRes(nbElement);

        thrust::transform(d_vecX.begin(), d_vecX.end(), d_vecY.begin(), d_vecRes.begin(),
                [=] __host__ __device__ (const float& x, const float& y){
                        return x * y;
                });

        // Transfer data to the host
        thrust::host_vector<float> h_vecRes = d_vecRes;

        std::cout << "x = " << h_vecX.front() << ", y = " << h_vecY.front() << ", res = " << h_vecRes.front() << std::endl;
        return 0;
}
```

Initialisation

Transfer Host to Device

Hadamard Product Thrust

```
int main(int argc, char** argv){
        size_t nbElement(500000000);

        thrust::host_vector<float> h_vecX(nbElement), h_vecY(nbElement);
        for(size_t i(0lu); i < nbElement; ++i){
                h_vecX[i] = i*19lu%11;
                h_vecY[i] = i*27lu%19;
        }
        // Transfer data to the device.
        thrust::device_vector<float> d_vecX = h_vecX, d_vecY = h_vecY, d_vecRes(nbElement);

        thrust::transform(d_vecX.begin(), d_vecX.end(), d_vecY.begin(), d_vecRes.begin(),
                [=] __host__ __device__ (const float& x, const float& y){
                        return x * y;
                });

        // Transfer data to the host
        thrust::host_vector<float> h_vecRes = d_vecRes;

        std::cout << "x = " << h_vecX.front() << ", y = " << h_vecY.front() << ", res = " << h_vecRes.front() << std::endl;
        return 0;
}
```

Initialisation

Transfer Host to Device

Compute Hadamard Product

Hadamard Product Thrust

```cpp
int main(int argc, char** argv){
        size_t nbElement(500000000);

        thrust::host_vector<float> h_vecX(nbElement), h_vecY(nbElement);
        for(size_t i(0lu); i < nbElement; ++i){
                h_vecX[i] = i*19lu%11;
                h_vecY[i] = i*27lu%19;
        }
        // Transfer data to the device.
        thrust::device_vector<float> d_vecX = h_vecX, d_vecY = h_vecY, d_vecRes(nbElement);

        thrust::transform(d_vecX.begin(), d_vecX.end(), d_vecY.begin(), d_vecRes.begin(),
                [=] __host__ __device__ (const float& x, const float& y){
                        return x * y;
                });

        // Transfer data to the host
        thrust::host_vector<float> h_vecRes = d_vecRes;

        std::cout << "x = " << h_vecX.front() << ", y = " << h_vecY.front() << ", res = " << h_vecRes.front() << std::endl;
        return 0;
}
```

Initialisation

Transfer Host to Device

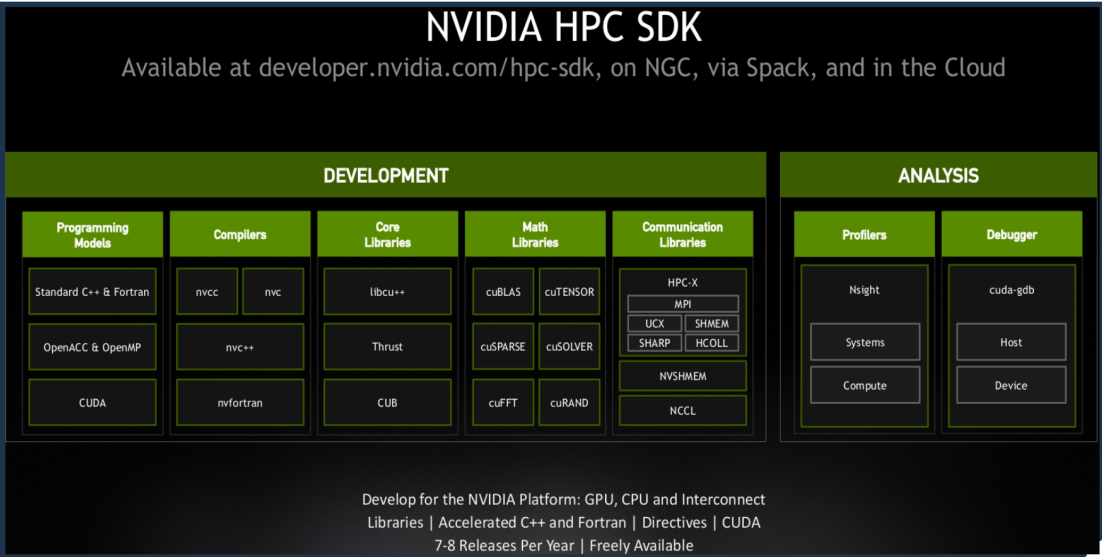
Compute Hadamard Product

Transfer Device to Host

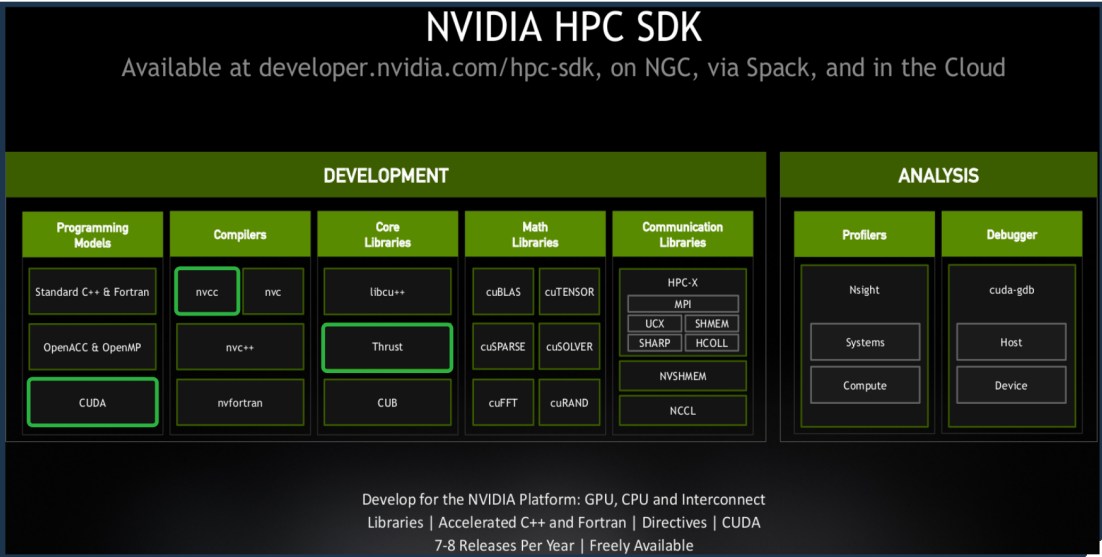
Hadamard Product Thrust

```cpp
#include <iostream>
#include <vector>

#include <thrust/transform.h>
#include <thrust/functional.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

int main(int argc, char** argv){
        size_t nbElement(500000000);

        thrust::host_vector<float> h_vecX(nbElement), h_vecY(nbElement);
        for(size_t i(0lu); i < nbElement; ++i){
                h_vecX[i] = i*19lu%11;
                h_vecY[i] = i*27lu%19;
        }
        // Transfer data to the device.
        thrust::device_vector<float> d_vecX = h_vecX, d_vecY = h_vecY, d_vecRes(nbElement);

        thrust::transform(d_vecX.begin(), d_vecX.end(), d_vecY.begin(), d_vecRes.begin(),
                [=] __host__ __device__ (const float& x, const float& y){
                        return x * y;
                });

        // Transfer data to the host
        thrust::host_vector<float> h_vecRes = d_vecRes;

        std::cout << "x = " << h_vecX.front() << ", y = " << h_vecY.front() << ", res = " << h_vecRes.front() << std::endl;
        return 0;
}
```

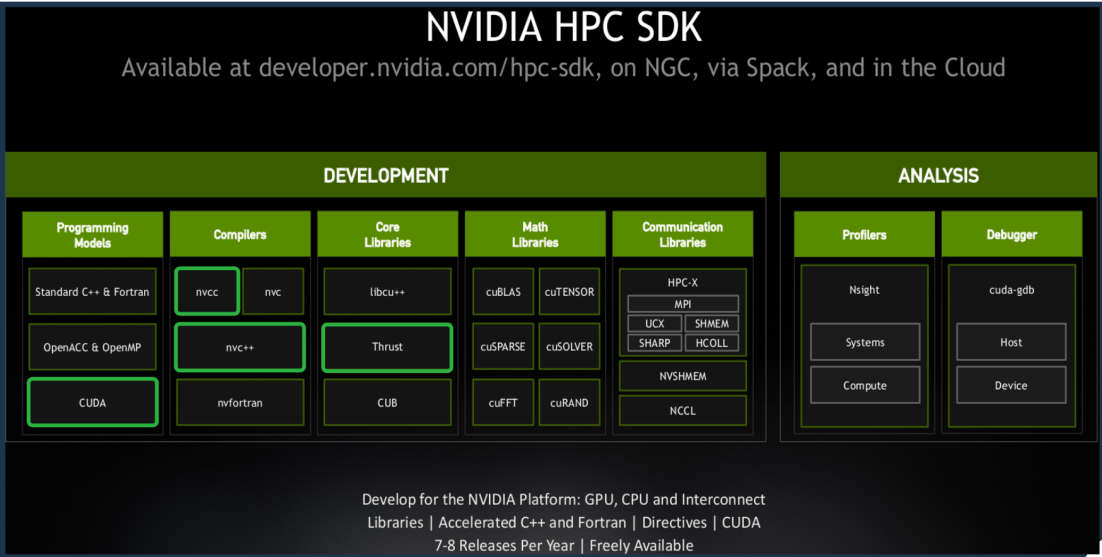
NVidia HPC SDK Introduction



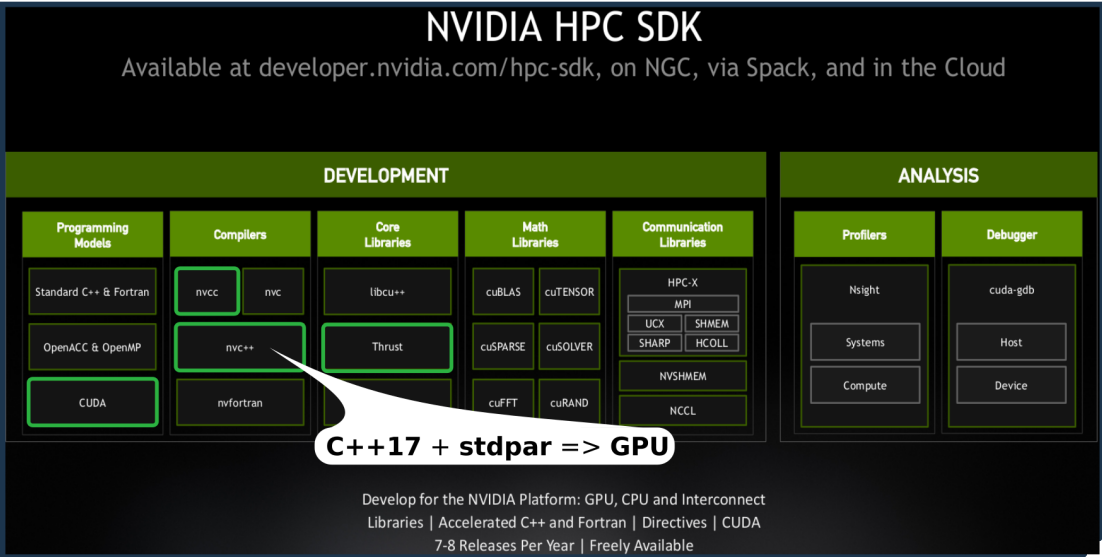
NVidia HPC SDK Introduction



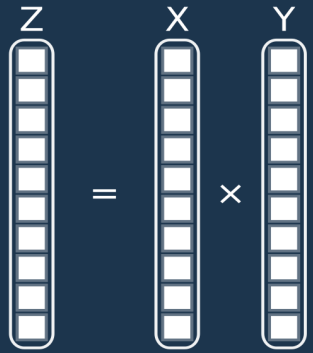
NVidia HPC SDK Introduction



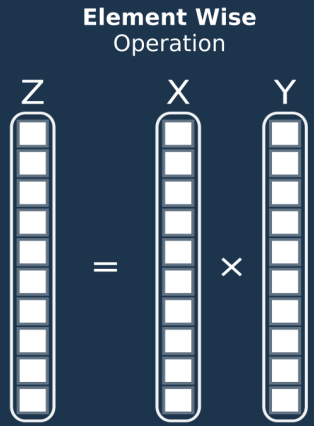
NVidia HPC SDK Introduction



Example : Hadamard Product



Example : Hadamard Product

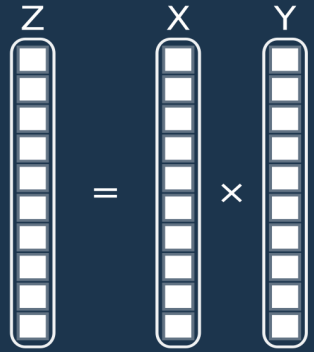


Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabResult[i] = tabX[i]*tabY[i];
}
```

Element Wise
Operation



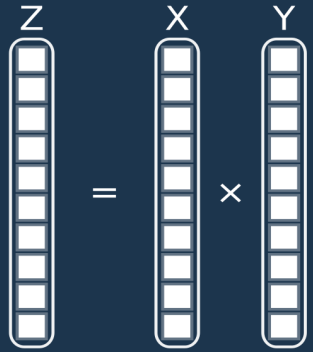
Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order **not** necessary

Element Wise Operation



Example : Hadamard Product

C++

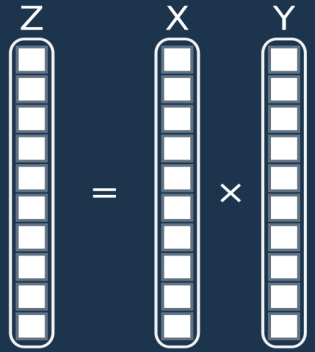
```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order **not** necessary

C++ Algorithm : std::transform

```
std::transform(std::begin(tabX), std::end(tabX),
»       std::begin(tabY), std::begin(tabRes),
»       [](float xi, float yi){ return xi * yi; });
```

Element Wise Operation



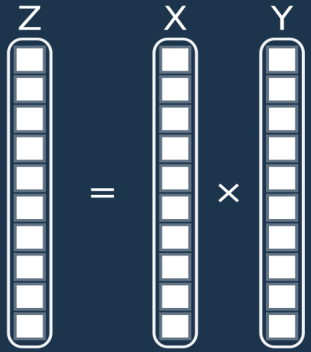
Example : Hadamard Product

C++

Explicit order **not** necessary

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabResult[i] = tabX[i]*tabY[i];
}
```

Element Wise Operation



C++ Algorithm : std::transform

```
std::transform(std::begin(tabX), std::end(tabX),
»       std::begin(tabY), std::begin(tabRes),
»       [](float xi, float yi){ return xi * yi; });
```

C++ 17 / C++ 20

```
std::transform(std::execution::par_unseq,
»       std::begin(tabX), std::end(tabX),
»       std::begin(tabY), std::begin(tabRes),
»       [](float xi, float yi){ return xi * yi; });
```

Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order **not** necessary

C++ Algorithm : std::transform

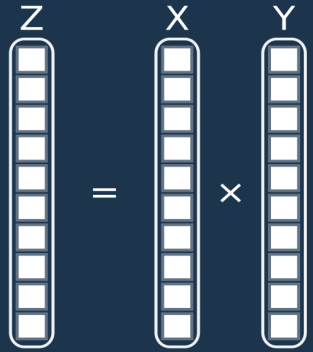
```
std::transform(std::begin(tabX), std::end(tabX),
»       std::begin(tabY), std::begin(tabRes),
»       [](float xi, float yi){ return xi * yi; });
```

C++ 17 / C++ 20

Execution Policy

```
std::transform(std::execution::par_unseq,
»       std::begin(tabX), std::end(tabX),
»       std::begin(tabY), std::begin(tabRes),
»       [](float xi, float yi){ return xi * yi; });
```

Element Wise Operation



Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order **not** necessary

C++ Algorithm : std::transform

```
std::transform(std::begin(tabX), std::end(tabX),
»       std::begin(tabY), std::begin(tabRes),
»       [](float xi, float yi){ return xi * yi; });
```

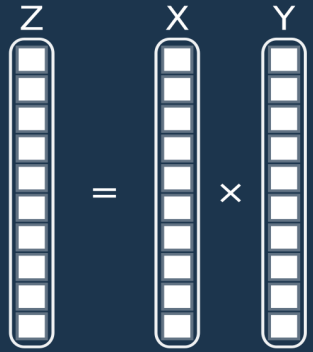
C++ 17 / C++ 20

Execution Policy

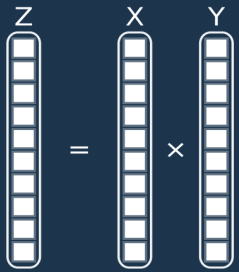
- seq
- unseq
- par
- par_unseq

```
std::transform(std::execution::par_unseq,
»       std::begin(tabX), std::end(tabX),
»       std::begin(tabY), std::begin(tabRes),
»       [](float xi, float yi){ return xi * yi; });
```

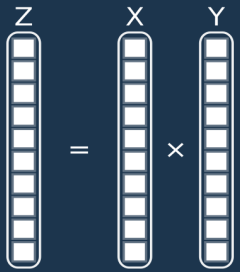
Element Wise Operation



Example : Hadamard Product



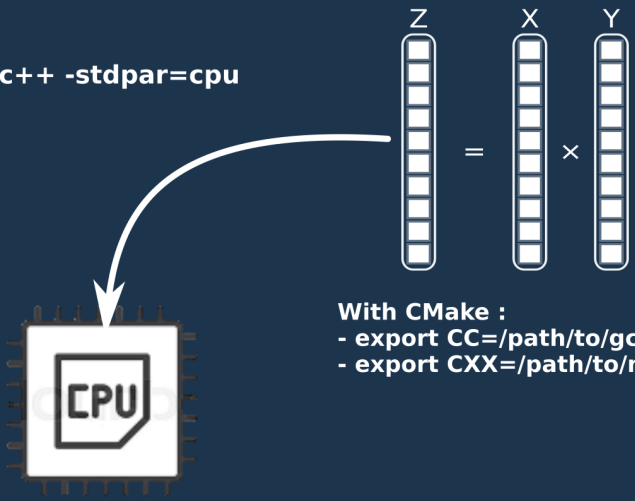
Example : Hadamard Product



With CMake :
- export CC=/path/to/gcc
- export CXX=/path/to/nvc++

Example : Hadamard Product

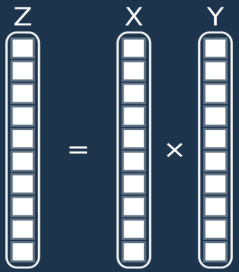
nvc++ -stdpar=cpu



With CMake :
- export CC=/path/to/gcc
- export CXX=/path/to/nvc++

Example : Hadamard Product

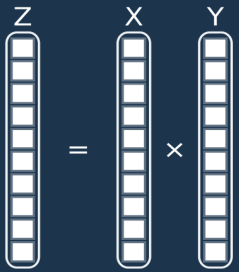
nvc++ -stdpar=cpu



With CMake :
- export CC=/path/to/gcc
- export CXX=/path/to/nvc++

Example : Hadamard Product

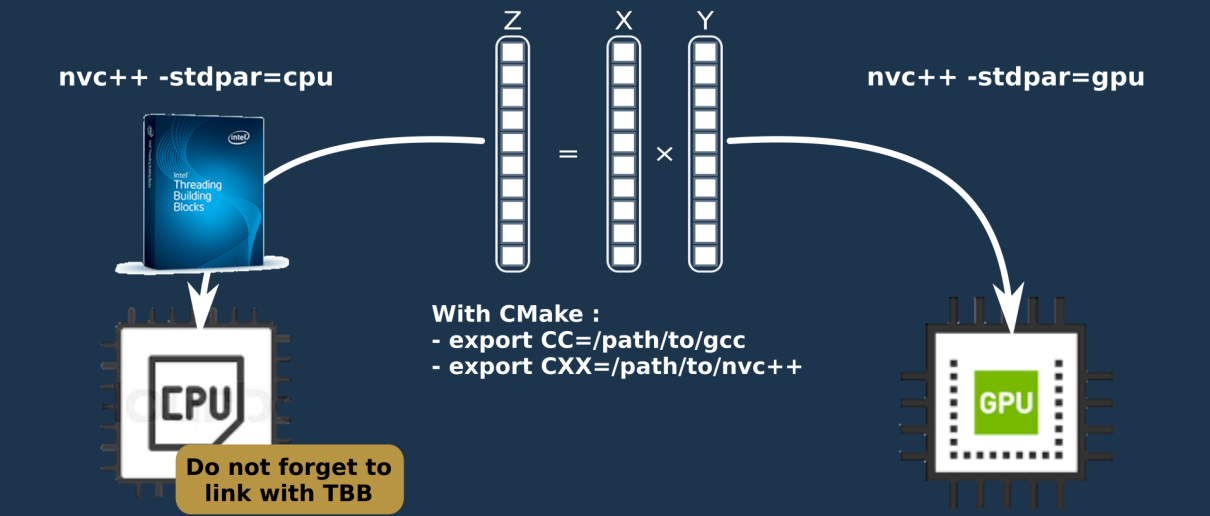
nvc++ -stdpar=cpu



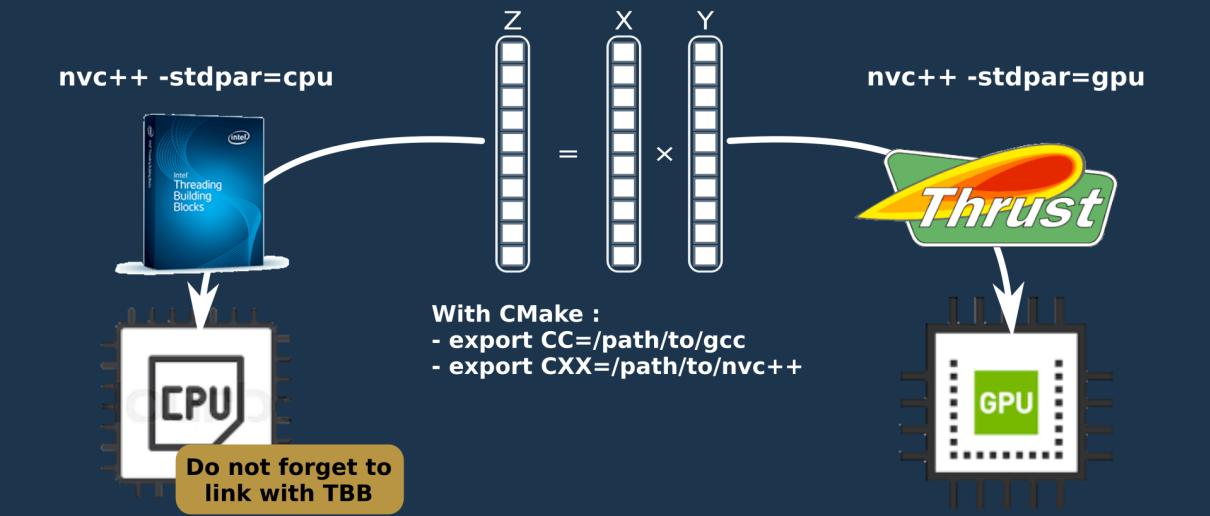
With CMake :
- **export CC=/path/to/gcc**
- **export CXX=/path/to/nvc++**

Do not forget to link with TBB

Example : Hadamard Product



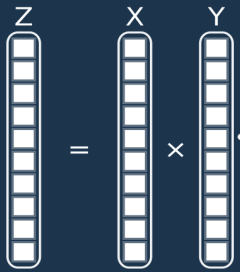
Example : Hadamard Product



Example : Hadamard Product

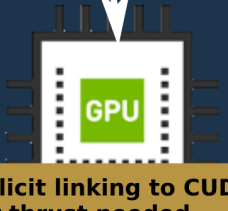
nvc++ -stdpar=cpu

Do not forget to link with TBB



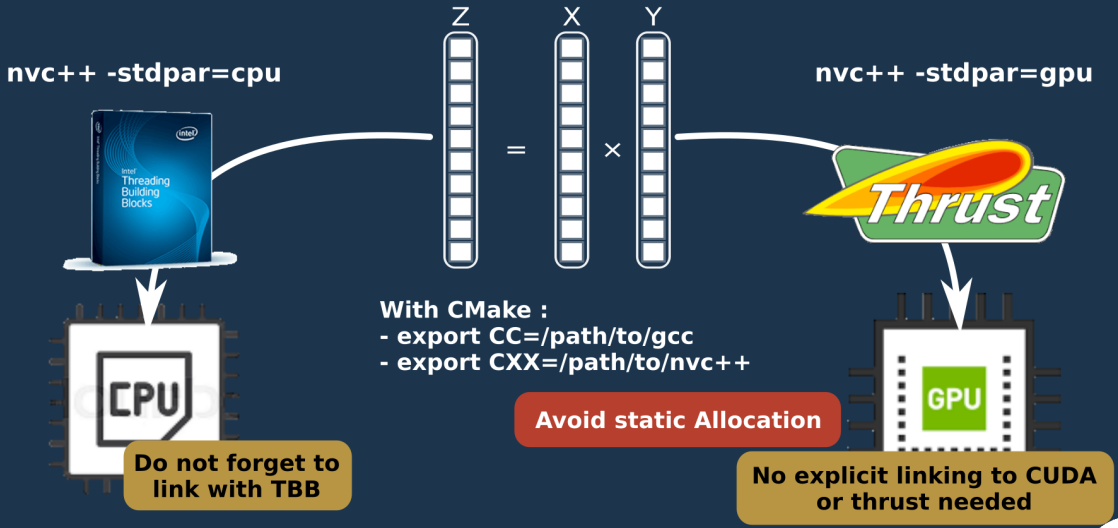
With CMake :
- export CC=/path/to/gcc
- export CXX=/path/to/nvc++

nvc++ -stdpar=gpu



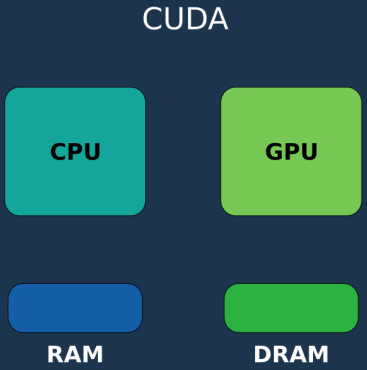
No explicit linking to CUDA or thrust needed

Example : Hadamard Product

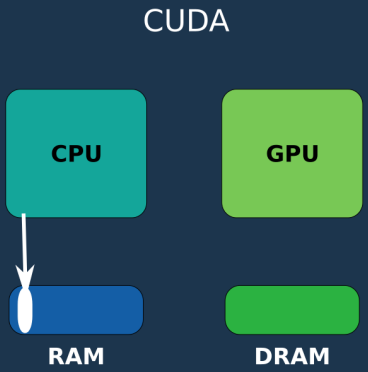


Memory Management

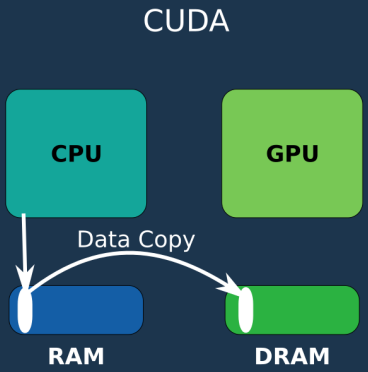
Memory Management



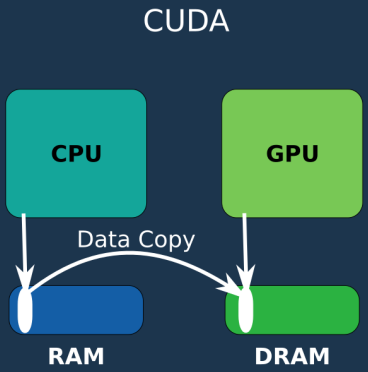
Memory Management



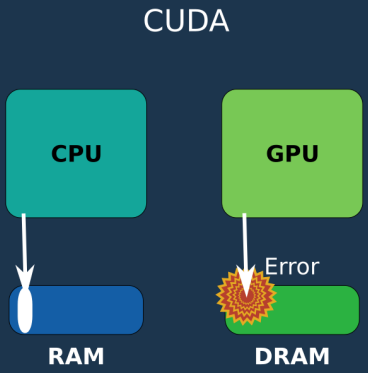
Memory Management



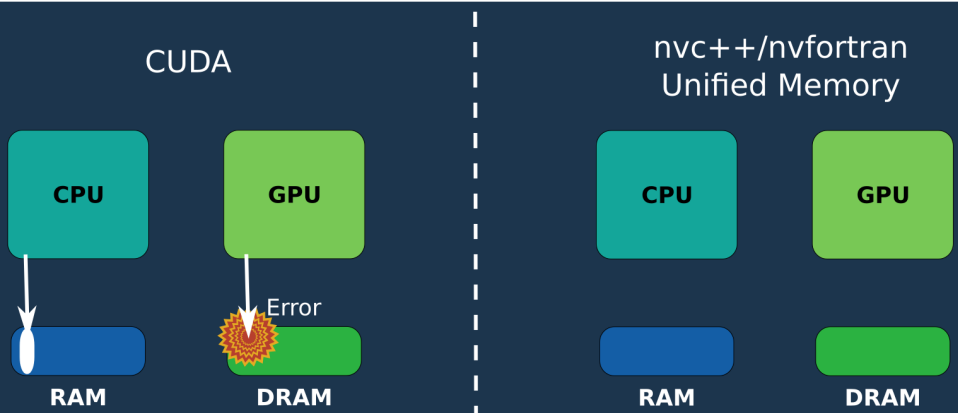
Memory Management



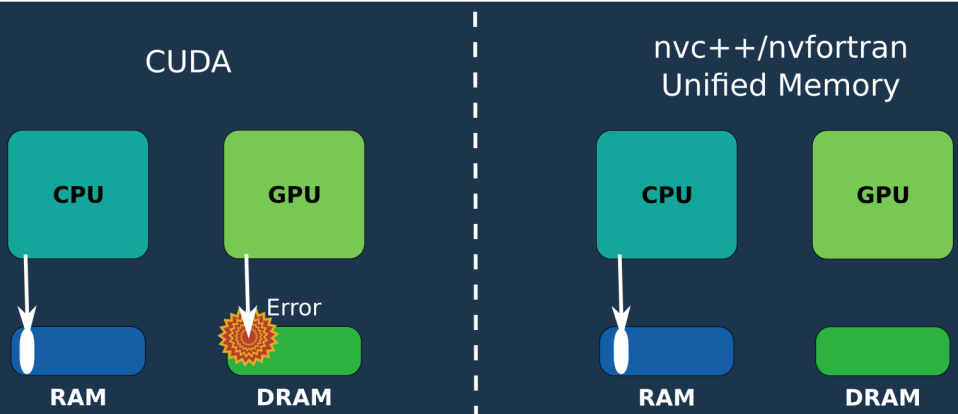
Memory Management



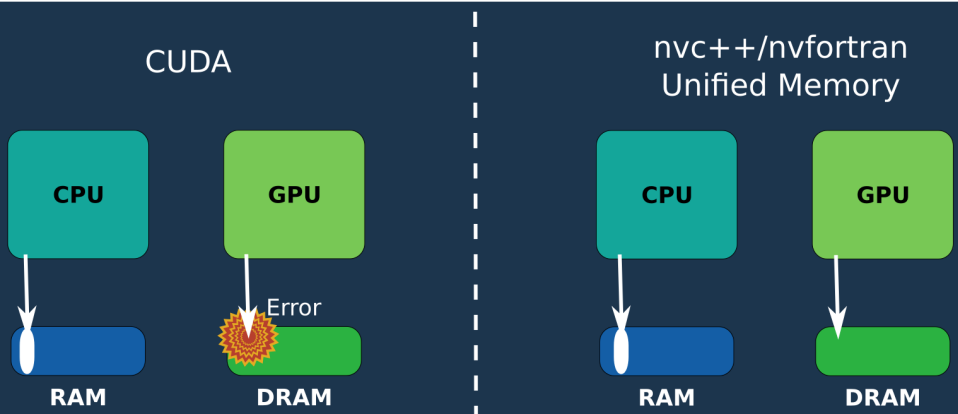
Memory Management



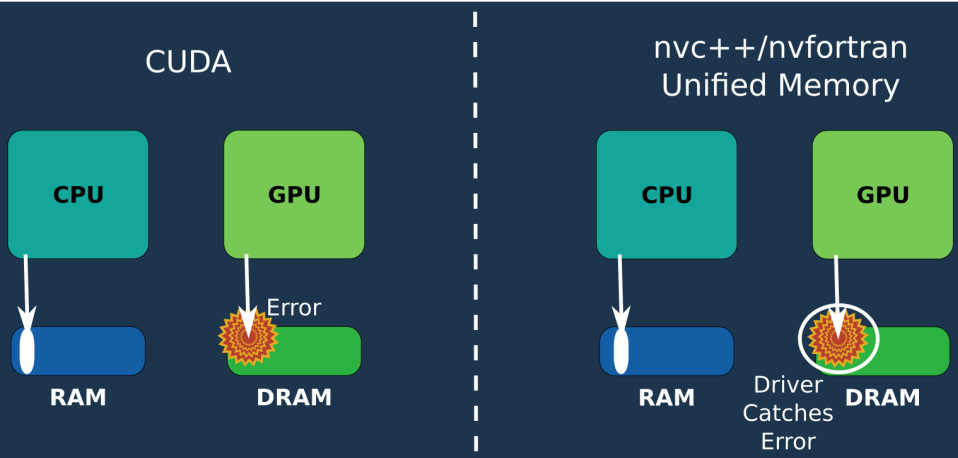
Memory Management



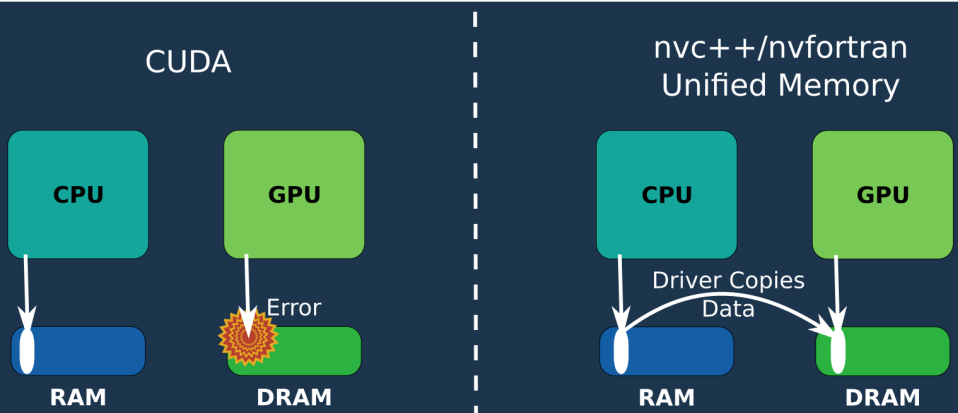
Memory Management



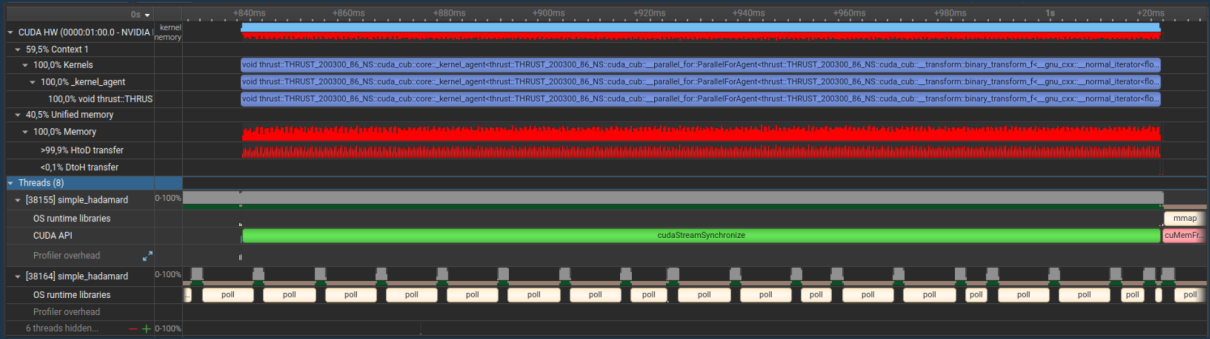
Memory Management



Memory Management



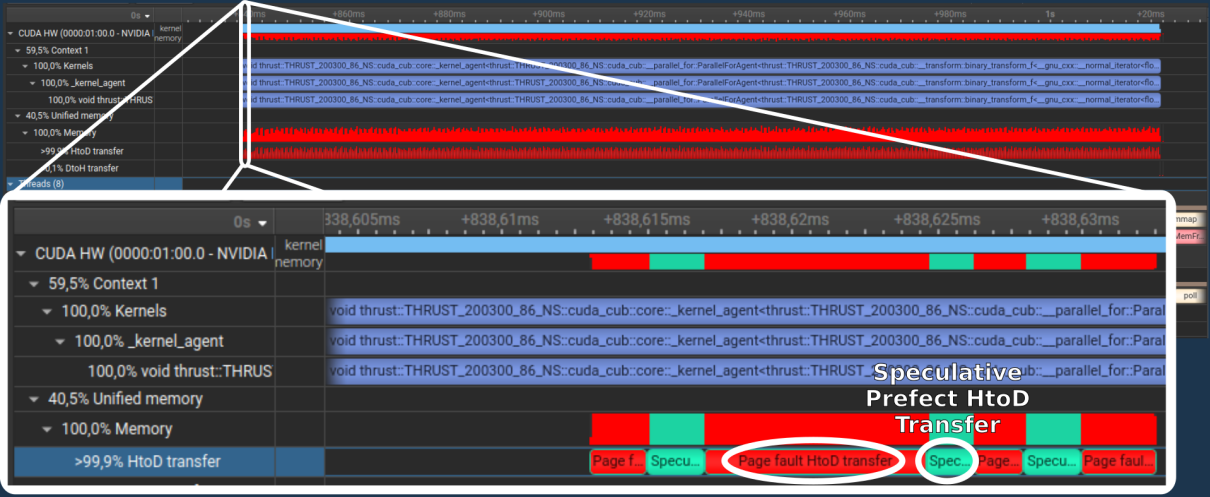
NSight : Hadamard Product



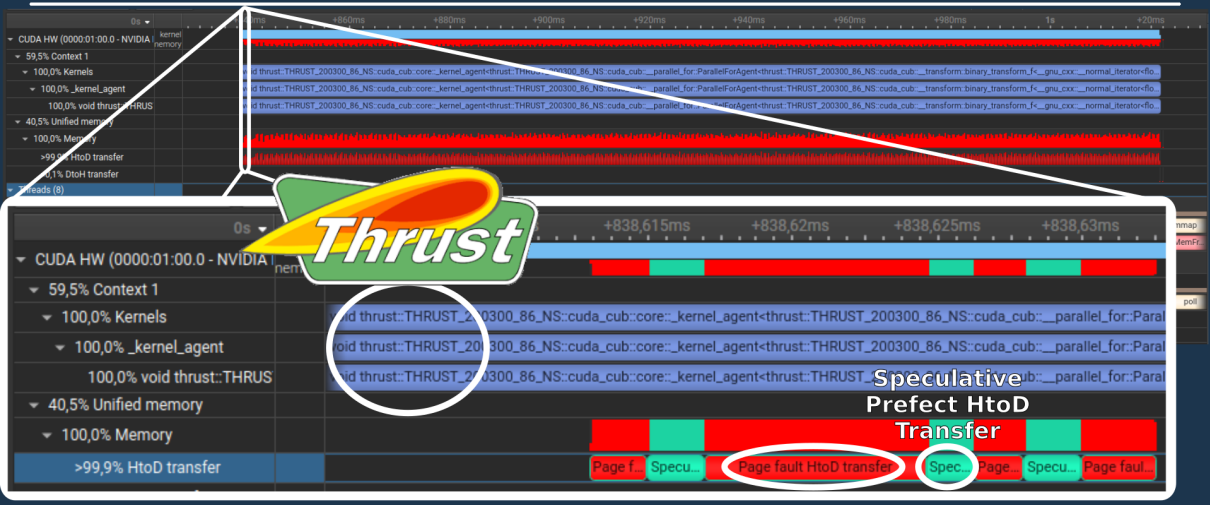
NSight : Hadamard Product

NSight : Hadamard Product

NSight : Hadamard Product



NSight : Hadamard Product



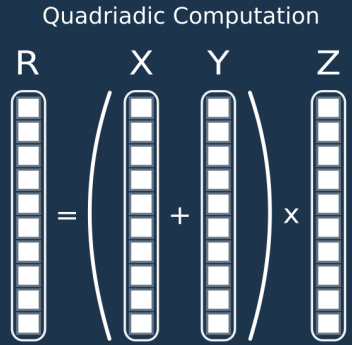
std::transform : triadic

std::transform : triadic

Triadic : **z = x + y**

std::transform : triadic

Triadic : **z = x + y**



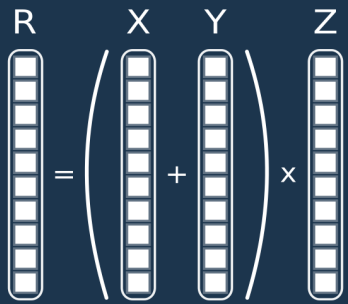
std::transform : triadic

Triadic : **z = x + y**

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

Quadriadic Computation



std::transform : triadic

Triadic : **z = x + y**

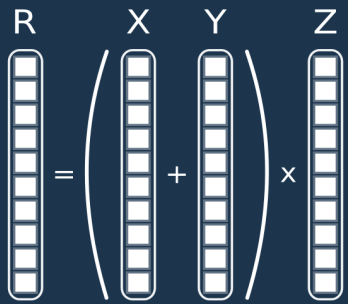
Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

C++ 17 / 20

```
std::transform(std::execution::par_unseq,
»       std::begin(vecIndex), std::end(vecIndex),
»       std::begin(vecX), std::begin(vecRes),
»       [=](int i, float x){
»       »       return (x + vecY[i]) * vecZ[i];
»       });
```

Quadriadic Computation



std::transform : triadic

Triadic : **z = x + y**

Classic C++

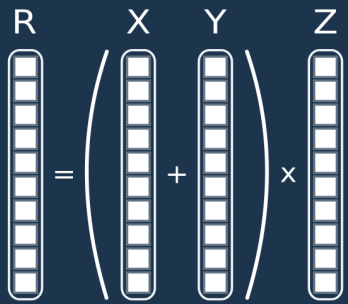
```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

C++ 17 / 20

```
std::transform(std::execution::par_unseq,
»       std::begin(vecIndex), std::end(vecIndex),
»       std::begin(vecX), std::begin(vecRes),
»       [=](int i, float x){
»       »       return (x + vecY[i]) * vecZ[i];
»       });
```

vecY, **vecZ** have to be **std::vector**

Quadriadic Computation



std::transform : triadic

Triadic : **z = x + y**

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

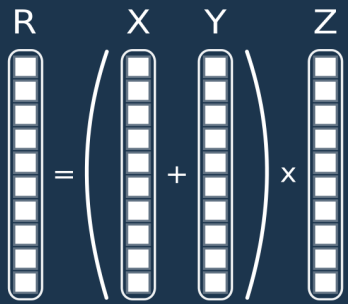
C++ 17 / 20

```
std::transform(std::execution::par_unseq,
»       std::begin(vecIndex), std::end(vecIndex),
»       std::begin(vecX), std::begin(vecRes),
»       [=](int i, float x){
»       »       return (x + vecY[i]) * vecZ[i];
»       });
```

vecY, **vecZ** have to be **std::vector**

Works well on GPU

Quadriadic Computation



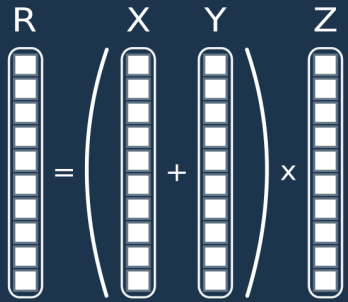
std::transform : triadic

Triadic : **z = x + y**

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

Quadriadic Computation



C++ 17 / 20

vecY, **vecZ** have to be **std::vector**

```
std::transform(std::execution::par_unseq,
»       std::begin(vecIndex), std::end(vecIndex),
»       std::begin(vecX), std::begin(vecRes),
»       [=](int i, float x){
»       »       return (x + vecY[i]) * vecZ[i];
»       });
```

Works well on GPU

Needs extra index table

std::transform : triadic

Triadic : **z = x + y**

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
»       tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

C++ 17 / 20

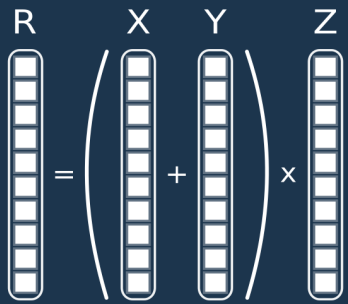
```
std::transform(std::execution::par_unseq,
»       std::begin(vecIndex), std::end(vecIndex),
»       std::begin(vecX), std::begin(vecRes),
»       [=](int i, float x){
»       »       return (x + vecY[i]) * vecZ[i];
»       });
```

vecY, **vecZ** have to be **std::vector**

Works well on GPU

Needs extra index table

Quadriadic Computation



Close to **CUDA Kernel**

C++23 Way

C++23 Way

Cartesian Product and **iota views**

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

C++23 Way

Cartesian Product and **iota views**

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

Multi-dimensional index

C++23 Way

Cartesian Product and **iota views**

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

No Index Allocation

Multi-dimensional index

C++23 Way

Cartesian Product and **iota views**

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

No Index Allocation

Multi-dimensional index

OK for broadcasting but we can do better

C++23 Way

Cartesian Product and **iota views**

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

No Index Allocation

Multi-dimensional index

OK for broadcasting but we can do better

Broadcast

C++23 Way

Cartesian Product and iota views

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

No Index Allocation

Multi-dimensional index

OK for broadcasting but we can do better

Broadcast


```
auto extentedVecB = std::views::join(std::views::repeat(vecB));
std::transform(std::execution::par, vecA.begin(), vecA.end(),
        std::begin(extentedVecB), vecResult.begin(),
        [=](float a, float b){
                return a * b;
        }
);
```

C++23 Way

Cartesian Product and iota views

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

No Index Allocation

Multi-dimensional index

OK for broadcasting but we can do better

Broadcast


```
auto extentedVecB = std::views::join(std::views::repeat(vecB));
std::transform(std::execution::par, vecA.begin(), vecA.end(),
        std::begin(extentedVecB), vecResult.begin(),
        [=](float a, float b){
                return a * b;
        }
);
```

Not vectorized yet

C++23 Way

Cartesian Product and iota views

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

No Index Allocation

Multi-dimensional index

OK for broadcasting but we can do better

Broadcast


```
auto extentedVecB = std::views::join(std::views::repeat(vecB));
std::transform(std::execution::par, vecA.begin(), vecA.end(),
        std::begin(extentedVecB), vecResult.begin(),
        [=](float a, float b){
                return a * b;
        }
);
```

Not vectorized yet

Not parallelized yet

C++23 Way

Cartesian Product and iota views

```
std::mdspan A{input, N, M};
std::mdspan B{output, M, N};
auto v = std::ranges::views::cartesian_product(
        std::ranges::views::iota(0, N),
        std::ranges::views::iota(0, M));

std::for_each(std::execution::par_unseq,
        begin(v), end(v),
        [=] (auto idx) {
                auto [i, j] = idx;
                B[j, i] = A[i, j];
        }
);
```

No Index Allocation

Multi-dimensional index

OK for broadcasting but we can do better

Broadcast


```
auto extentedVecB = std::views::join(std::views::repeat(vecB));
std::transform(std::execution::par, vecA.begin(), vecA.end(),
        std::begin(extentedVecB), vecResult.begin(),
        [=](float a, float b){
                return a * b;
        }
);
```

Not vectorized yet

Not parallelized yet

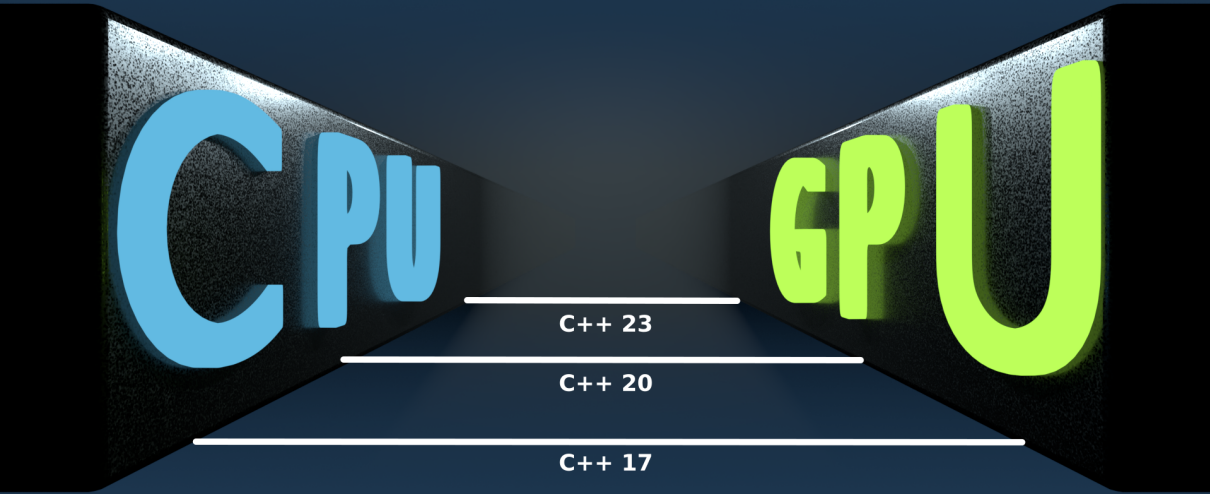
But only with G++ 13 For now

C++ Evolution

C++ Evolution

C++ Evolution

C++ Evolution



C++ Evolution

Conclusion

- Good performances on **GPU**s
  - With **nvcc** (**CUDA**)
  - With **nvc++** (**C++17** / **C++20**)
- Tests with **HPC SDK 24.3**
- Compiler **nvc++** powerful and easy to use with **C++17** and **C++20**
  - **No explicit linking**
  - Automatic **GPU** Targeting or with **CUDA_VISIBLE_DEVICES**
  - **Avoid static allocation**
- Warning about industrial software
  - Will to drive for update
  - Old **GPU**s become **obsolete** :
    - **nvc++** : compute capabilities ≥ 6 (no **K80**)
    - **nvcc** : compute capabilities ≥ 3.5 (no **K80** in **CUDA 12**)
  - Need to save binaries to ensure long usability of **GPU**s