Multi-dimensional grids/blocks, coalescence

Oguz Kaya

Maˆıtre de Conf'erences
Universit'e Paris-Saclay and the LRI ParSys team, Orsay, France

université
PARIS-SACLAY

Objectives

- Use the indexing of grids/blocks 2 or 3 dims Realize
- "fast" access the global memory of the GPU First TP on
- matrix multiplication

Outline

Outline

Multiplication of an array 3D by blocks

Recapultative of the example of the multiplication of an array by blocks

```
#include <cstdio>
#include "cuda.h"

#define N 1024
float A[N];
float c = 2.0;

__device__ float dA[N];

__global__ void multiplyArray(int n, float c)
{
  int i = blockIdx.x;
  dA[i] *= c;
}

int main(int argc, char **argv)
{
  // Initialization
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  multiplyArray<<<N, 1>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  return 0;
}
```

- Multiply a 1D array by a grid of blocks.
- Each block multiplies 1 element. Run
- the kernel with N blocks.
- **Question:** What would happen if the etait plutôt $A[N][N]$?

Multiplying a 2D array by blocks

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x / n;
  int j = blockIdx.x % n;
  dA[i][j] *= c;
}

int main(int argc, char **argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  multiplyArray2D<<<N * N, 1>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Each block multiplies 1 element.
- You have to throw $N2$ blocks in total.
- Every N consecutive blocks multiply a line of A.

A division and a modulus to find i and j

Multiplication of a 2D table by 2D blocks

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  dA[blockIdx.x][blockIdx.y] *= c;
}
int main(int argc, char **argv)
{
  for (int i = 0; i < N; i++) {   // Initialization
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
       cudaMemcpyHostToDevice);
  dim3 dimGrid;
  dimGrid.x = N;
  dimGrid.y = N;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, 1>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
       cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- **dim3** defines a 3D indexing topology (**dim3.{x,y,z}**).
- Set **dim3.z = 1** for 2D.
- The total number of blocks is equal the multiplication of dimensions.
- No division or modulus to find i and j

Multiplication of a 2D table by 2D blocks

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x;
  int j = blockIdx.y * blockDim.x + threadIdx.x;
  if (j < n) { dA[i][j] *= c; }
}

int main(int argc, char **argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  int blockSize = 1024;
  dim3 dimGrid;
  dimGrid.x = N;
  dimGrid.y = N / blockSize;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, blockSize>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Each block multiplies blockSize elements. Each thread multiplies 1 element.
- Threads work on consecutive elements in a **line**.
- You have to run *N2/blockSize* blocks in total. Each block multiplies a subline of *A*.
- Check the overflow of the array.

Multiplication of a 2D table by 2D blocks

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y;
  if (i < n) { dA[i][j] *= c; }
}

int main(int argc, char **argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
       cudaMemcpyHostToDevice);
  int blockSize = 1024;
  dim3 dimGrid;
  dimGrid.x = N / blockSize
  ; dimGrid.y = N;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, blockSize>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
       cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Each thread multiplies 1 element.
- It is necessary to launch *N2/blockSize* blocks
- in total. Threads work on consecutive elements
- in a **column**.
- Each block multiplies a sub-column of *A*.
- Which is better (subline vs. subcolumn)?
- What happens if *A* has few rows/columns?

Multiplication of a 2D table by 2D blocks

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

# define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int blockDimSqrt = (int) sqrt((float) blockDim.x);
  int i = blockIdx.x * blockDimSqrt + threadIdx.x / blockDimSqrt;
  int j = blockIdx.y * blockDimSqrt + threadIdx.x % blockDimSqrt;
  if (i < n && j < n) { dA[i][j] *= c; }
}

int main(int argc, char **argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  int blockSize = 1024;
  dim3 dimGrid;
  dimGrid.x = N / 32;
  dimGrid.y = N / 32;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, blockSize>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Distribute 1024 threads in 2D.
- Each block takes care of one

  32×32 sub-matrix.
- Threads consecutifs is working on a **subline**.
- Finding i and j does not require division and modulus.

Multiplication of a 2D table by 2D blocks

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

# define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int blockDimSqrt = (int) sqrt((float) blockDim.x);
  int i = blockIdx.x * blockDimSqrt + threadIdx.x % blockDimSqrt
  ; int j = blockIdx.y * blockDimSqrt + threadIdx.x / blockDimSqrt
  ; if (i < n && j < n) { dA[i][j] *= c; }
}

int main(int argc, char ** argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  int blockSize = 1024;
  dim3 dimGrid;
  dimGrid.x = N / 32;
  dimGrid.y = N / 32;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, blockSize>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Distribute 1024 threads in 2D.
- Each block takes care of a 32×32 sub-matrix.
- Threads consecutifs is working on a **sub-column**.
- Finding i and j does not require division and modulus.
- Which one is better (subline vs. subcolumn access)?

Multiplication of a 2D table by 2D blocks

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < n && j < n) { dA[i][j] *= c; }
}

int main(int argc, char ** argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  dim3 dimBlock;
  dimBlock.x = 32;
  dimBlock.y = 32;
  dimBlock.z = 1;
  dim3 dimGrid;
  dimGrid.x = N / 32;
  dimGrid.y = N / 32;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, dimBlock>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Using a **dim3** for 2D indexing of threads.
- Consecutive threads in **threadIdx.x** are executed in a **warp** (then in **threadIdx.y**, then into **threadIdx.z**).
- So each warp has access to a **sub-column** of A.

Multiplication of a 2D table by 2D blocks

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N]
; float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x * blockDim.y + threadIdx.y
  ; int j = blockIdx.y * blockDim.x + threadIdx.
  x; if (i < n && j < n) { dA[i][j] *= c; }
}

int main(int argc, char ** argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  dim3 dimBlock;
  dimBlock.x = 32;
  dimBlock.y = 32;
  dimBlock.z = 1;
  dim3 dimGrid;
  dimGrid.x = N / 32;
  dimGrid.y = N / 32;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, dimBlock>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Use a **dim3** for 2D thread indexing.
- Consecutive threads in **threadIdx.x** are executed in a **warp** (then in **threadIdx.y**, then into **threadIdx.z**).
- So each warp has access to a **subline** of *A*.
- Which one is better?

Grid and block size limits

For a grid, you must use

- **dim3.x ≤** 2^{31} - 1
- **dim3.y ≤** 65535
- **dim3.z ≤** 65535

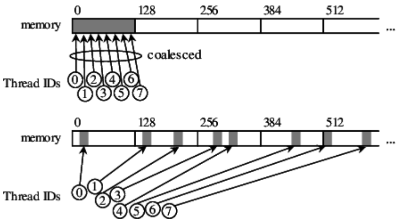
For a block, you must use

- **dim3.x ≤** 1024
- **dim3.y ≤** 1024
- **dim3.z ≤** 64
- Total number of threads ≤ 1024

Outline

Coalescence

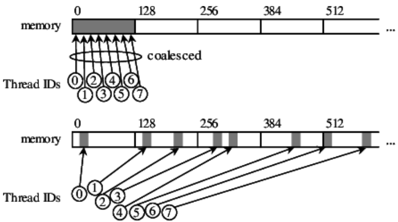
This is access the global memory of threads in a warp.



- Threads in a warp execute instructions **synchronously**.
- Each memory access is also processed **synchronously**.
- If the threads access consecutive elements in the memory, this only requires the opening of one line (so 1 access).

Coalescence (cont.)

This is access the global memory of threads in a warp.



- If there are jumps, each line touched will be read.
  - At limit, 32 lines read for one access. Most
  - elements in the line are unused.
- **Rule:** Design the kernel so that accesses are contiguous in **threadIdx.x**.

Example: Multiplication of a 2D array

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x;
  int j = blockIdx.y * blockDim.x + threadIdx.x;
  if (j < n) { dA[i][j] *= c; }
}

int main(int argc, char **argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  int blockSize = 1024;
  dim3 dimGrid;
  dimGrid.x = N;
  dimGrid.y = N / blockSize;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, blockSize>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Is it **coalescing**?
- Yes! The matrix is stored in rows, **threadIdx.x** aligne on lines.

Example: Multiplication of a 2D array

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y;
  if (i < n) { dA[i][j] *= c; }
}

int main(int argc, char **argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  int blockSize = 1024;
  dim3 dimGrid;
  dimGrid.x = N / blockSize
  ; dimGrid.y = N;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, blockSize>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Is it **coalescing**?
- No! The matrix is stored in rows, **threadIdx.x** aligné on columns.
- 32 l'ectures mèmoires n'ecessaires pour chaque accès mèmoire d'un warp.

Example: Multiplication of a 2D array

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x; int j = blockIdx.y * blockDim.y + threadIdx.y; if (i < n && j < n) { dA[i][j] *= c; }
}

int main(int argc, char **argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  dim3 dimBlock;
  dimBlock.x = 32;
  dimBlock.y = 32; dimBlock.z = 1; dim3 dimGrid;
  dimGrid.x = N / 32;
  dimGrid.y = N / 32;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, dimBlock>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
```

- Is it **coalescing**?
- No! The matrix is stored in rows, **threadIdx.x** alignè on columns.
- 32 l'ectures mèmoires n'ecessaires pour chaque accès mèmoire d'un warp.

Example: Multiplication of a 2D array

Multiply each element of an array $A[N][N]$ by a scalar c

```
#include <cstdio>
#include "cuda.h"

#define N 2048
float A[N][N];
float c = 2.0;

__device__ float dA[N][N];

__global__ void multiplyArray2D(int n, float c)
{
  int i = blockIdx.x * blockDim.y + threadIdx.
  y; int j = blockIdx.y * blockDim.x + threadId
  x.x; if (i < n && j < n) { dA[i][j] *= c; }
}

int main(int argc, char ** argv)
{
  // Initialization
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copy the table to the GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  dim3 dimBlock;
  dimBlock.x = 32;
  dimBlock.y = 32; dim
  Block.z = 1; dim3 di
  mGrid;
  dimGrid.x = N / 32;
  dimGrid.y = N / 32;
  dimGrid.z = 1;
  multiplyArray2D<<<dimGrid, dimBlock>>>(N, c);
  // Copy the array multiplies to the CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
```

- Is it **coalescing**?
- Yes! The matrix is stored in rows, **threadIdx.x** alignè on lines.

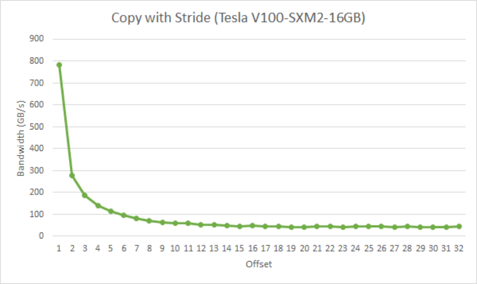
Example: Access an array with jumps

```
__device__ float dA[N];

__global__ void stridedAccess(int stride)
{
  float f = dA[threadIdx.x * stride];
  // ...
}
```

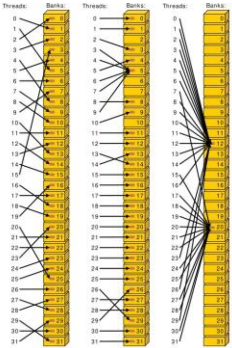
- How the performance evolve in relation to the **stride**?
- For **stride** = 1, the ecture of a 128 bytes line.
- For **stride** = 2, the reading of two lines of 128 bytes (half unused)
- ...
- For **stride** = 32, the reading of 32 lines of 128 bytes (31/32 unused)

Performance of the access to an array with jumps



- The effective bandwidth drops very quickly with jumps.

Coalescence rules



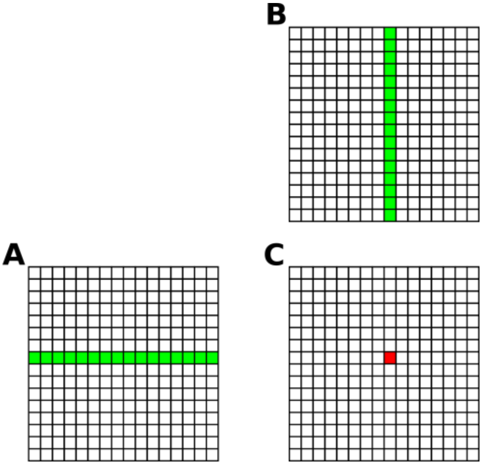
Left
Conflict-free access via random permutation.
Middle
Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank
Right
Conflict-free broadcast access (threads access the same word within a bank).

- Threads in a warp access the **même case mémoire** = good performance (yet bande passent toujours gaspillée).
- Threads in a warp access the **même line in memory** in an **aléatoire** order = still good performance in new architectures.
- If coalescence is difficult to do, **shared memory** could être useful (coming soon).

Outline

Multiplication of matrices

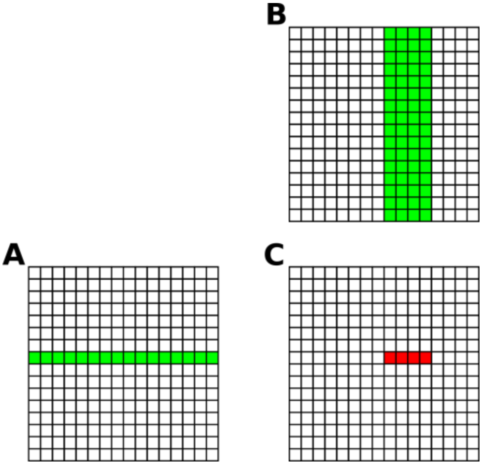
Let A, B, C be matrices of size $N \times N$.



- The multiplication $C = AB$ corresponds to the calculation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- First kernel: Create a block for each $C[i][j]$.

Multiplication of matrices

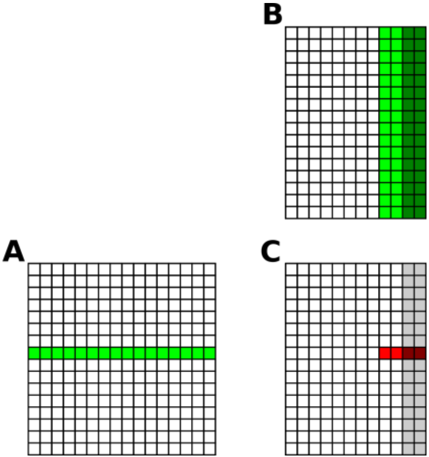
Let A, B, C be matrices of size $N \times N$.



- The multiplication $C = AB$ corresponds to the calculation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Second kernel: Use P threads per block, each block computes P main elements of a line of C ($P = 4$ here).
- Assume that P divides N.

Multiplication of matrices

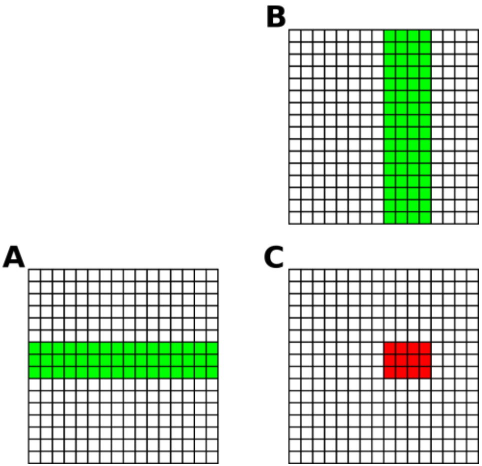
Let A, B, C be matrices of size $N \times N$.



- The multiplication $C = AB$ corresponds to the calculation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Third kernel: Use P threads per block, each block computes P [illegible]main elements of a line of C ($P = 4$ here).
- Assume that P does not divide N.

Multiplication of matrices

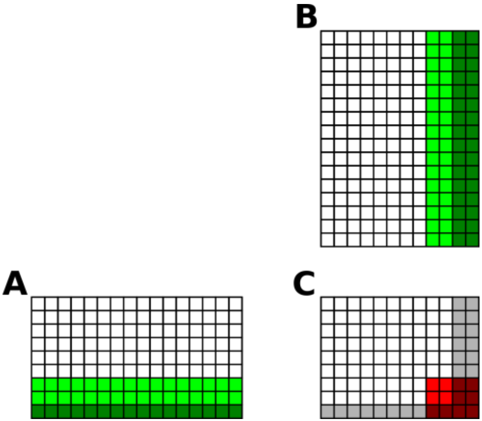
Let A, B, C be matrices of size $N \times N$.



- The multiplication $C = AB$ corresponds to the calculation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Fourth kernel: Use PQ threads per block, each block computes $P \times Q$ The following is a list of components of a sub-block of C ($P = 4$ and $Q = 3$ here).
- Assume that P and Q divide N.

Multiplication of matrices

Let A, B, C be matrices of size $N \times N$.



- The multiplication $C = AB$ corresponds to the calculation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Fifth kernel: Use PQ threads per block, each block computes $P \times Q$ consecutive elements of a sub-block of C ($P = 4$ and $Q = 3$ here).
- Assume that neither P nor Q divides N.

Contact

Oguz Kaya
Université Paris-Saclay and LRI, Paris, France
oguz.kaya@lri.com
www.oguzkaya.com