Multi-dimensional grids/blocks and coalescence

Oguz Kaya

Assistant Professor
Université Paris-Saclay, Gif-sur-Yvette, France

Objectives

- Use 2D and 3D block indices
- Perform “fast” memory access to the global GPU memory
- Programming exercise on GPU matrix multiplication

Outline

Outline

Multiplication of a 1D array by blocks

Summary of the multiplication example of an 1D 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)
{
  // Initialisation
  for (int i = 0; i < N; i++) { A[i] = i; }
  // Copier le tableau vers le GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  multiplyArray<<<N, 1>>>(N, c);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  return 0;
}
```

- Multiply each element of a 1D array with constant c using a grid of blocks
- Each block multiplies 1 element
- Execute the kernel using N blocks
- **Question:** What would we do if the array were 2D, i.e. $A[N][N]$?

Multiplication of a 2D array by blocks

```
#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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le GPU
  cudaMemcpyToSymbol(dA, A, N * N * sizeof(float), 0,
      cudaMemcpyHostToDevice);
  multiplyArray2D<<<N * N, 1>>>(N, c);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Multiply each element of $A[N][N]$ by a constant c
- Each block multiplies 1 element
- We need N^2 blocks in total.
- Each group of N consecutive blocks multiply a row of A
- With division and modulus, we can find indices i, j for $A[i][j]$ to be multiplied by each block

Multiplication of a 2D array by a grid of 2D blocks

```
#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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Multiply each element of $A[N][N]$ by a constant c
- Each block multiplies 1 element
- **dim3** defines a 3D index topology for blocks in a grid (**dim3.{x,y,z}**).
- Use **dim3.x = dim3.y = N** and **dim3.z = 1** pour 2D.
- Total number of blocks is **dim3.x * dim3.y * dim3.z**
- No need for division or modulus tricks to find i and j

Multiplication of a 2D array by 2D blocks and 1D threads

```
#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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Multiply each element of $A[N][N]$ by a constant c
- Each block multiplies **blockSize** elements
- Each thread multiplies 1 element
- Threads in a block work on **blockSize** consecutive elements in a **row** of A.
- Need to launch $N^2/blockSize$ blocks in total.
- Each block multiplies a part of a row of A
- Need to make sure not to make out-of-bounds memory accesses for the last threads

Multiplication of a 2D array by 2D blocks and 1D threads

```
#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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Each thread multiplies 1 element
- Need to launch $N^2/blockSize$ blocks in total.
- Threads in a block work on **blockSize** consecutive elements in a **column** of A.
- Need to launch $N^2/blockSize$ blocks in total.
- Each block multiplies a part of a column of A
- Need to make sure not to make out-of-bounds memory accesses for the last threads
- Which one is better (row-based or col-based)?
- What happens if A has few rows/columns?

Multiplication of a 2D array by 2D blocks and 1D threads

```
#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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Organize **blockSize = 1024** threads in 2D
- Each block works on a submatrix of size 32×32
- Consecutive threads work on the same **row**
- Find i and j with division and modulus

Multiplication of a 2D array by 2D blocks and 1D threads

```
#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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Organize **blockSize = 1024** threads in 2D
- Each block works on a submatrix of size 32×32
- Consecutive threads work on the same **column**
- Find i and j with division and modulus
- Which one is better (row-based or col-based)?

Multiplication of a 2D array by 2D blocks and 1D threads

```
#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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Use a **dim3** for 2D indexing of threads, with **dim3.x = dim3.y = 32** and **dim3.z = 1**
- Threads with consecutive **threadIdx.x** are put in the same warp (then using **threadIdx.y**, then **threadIdx.z**)
- Therefore, each warp touches a **column** of A

Multiplication of a 2D array by 2D blocks and 1D threads

```
#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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Use a **dim3** for 2D indexing of threads, with **dim3.x = dim3.y = 32** and **dim3.z = 1**
- Threads with consecutive **threadIdx.x** are put in the same warp (then using **threadIdx.y**, then **threadIdx.z**)
- Therefore, each warp touches a **row** of A
- Which one is better (row-based or col-based)?

Dimension limits for grids and blocks

For a grid, need to have

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

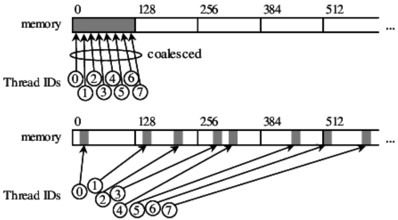
For a block, need to have

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

Outline

Coalescence

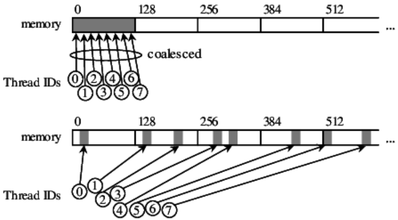
Coalescence pertains to accesses to the main memory from threads within a warp.



- Threads within a warp execute instructions **synchronously**
- Each memory access is equally treated **synchronously**
- If threads access to consecutive elements in the memory, it requires reading/writing one memory line (thus performs 1 memory access)

Coalescence (cont.)

Il s'agit d'accès à la mémoire globale des threads dans un warp.



- If the access is scattered, each touched line will be read
  - In the worst case, 32 lines might be read for a single warp access
  - Most read elements will not be used; bandwidth wasted.
- **Rule:** Design the kernel so that the accesses are contiguous on **threadIdx.x** (and then **threadIdx.y**, then **threadIdx.z**)

Example: Multiplication of a 2D array

Multiply each element of an array $A[N][N]$ with 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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Is this **coalescent**?
- Yes! Matrix is stored by rows, **threadIdx.x** aligned with rows

Example: Multiplication of a 2D array

Multiply each element of an array $A[N][N]$ with 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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Is this **coalescent**?
- No! Matrix is stored by rows, **threadIdx.x** aligned with columns
- 32 lines will be read for each memory access performed by a warp

Example: Multiplication of a 2D array

Multiply each element of an array $A[N][N]$ with 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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
```

- Is this **coalescent**?
- No! Matrix is stored by rows, **threadIdx.x** is aligned with columns.
- 32 lines will be read for each memory access performed by a warp

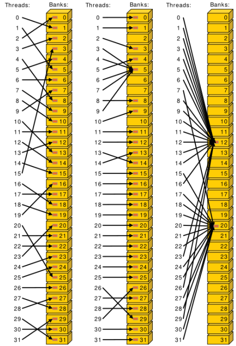
Example: Multiplication of a 2D array

Multiply each element of an array $A[N][N]$ with 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)
{
  // Initialisation
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) { A[i][j] = i + j; }
  }
  // Copier le tableau vers le 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);
  // Recopier le tableau multiplie vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
```

- Is this **coalescent**?
- Yes! Matrix is stored by rows, **threadIdx.x** is aligned with rows

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).

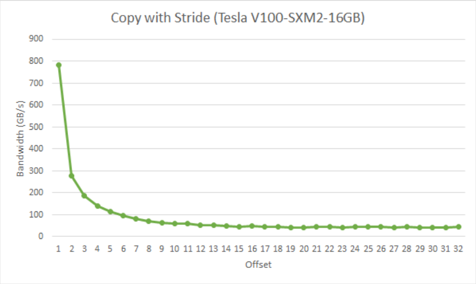
- Threads in a warp accessing the same memory slot = good performance (however, bandwidth is still potentially wasted).
- Threads in a warp accessing the same **memory line** in a **random** order = still good performance in new architectures (Volta and later).
- If coalescent access is difficult to do, **shared memory** might be useful (we will see soon).

Example: Strided memory access to an array

```
__device__ float dA[N];

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

- How does performance evolve in terms of **stride**?
- For **stride** = 1, reading a single line of 128 bytes.
- For **stride** = 2, reading two lines of 128 bytes (half of which is unused).
- ...
- For **stride** = 32, reading 32 lines of 128 bytes (31/32 of which is unused).

Example: Strided memory access to an array (cont.)

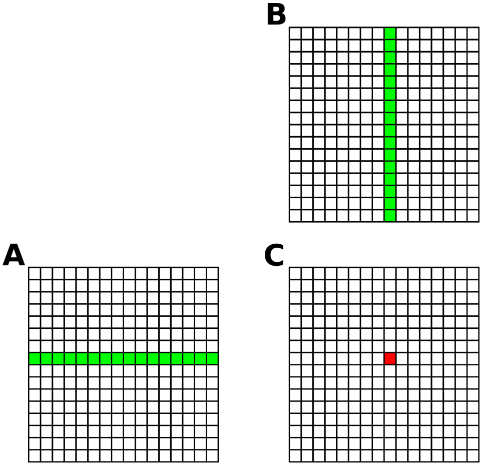


- Effective bandwidth falls rapidly.

Outline

Matrix multiplication

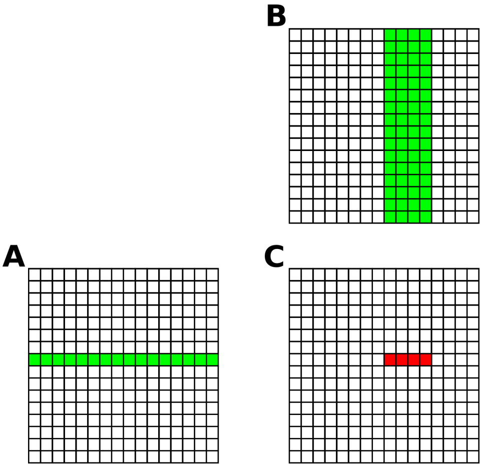
Let A, B, C be $N \times N$ matrices.



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

Matrix multiplication

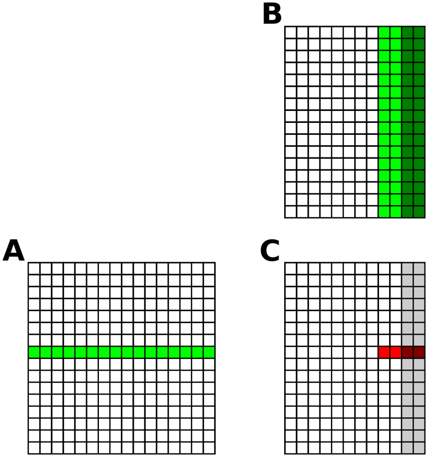
Let A, B, C be $N \times N$ matrices.



- The multiplication $C = AB$ corresponds to the computation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Second kernel: Use P threads per block, each block computing P consecutive elements of a row of C ($P = 4$ in this figure).
- Suppose that N is divisible by P.

Matrix multiplication

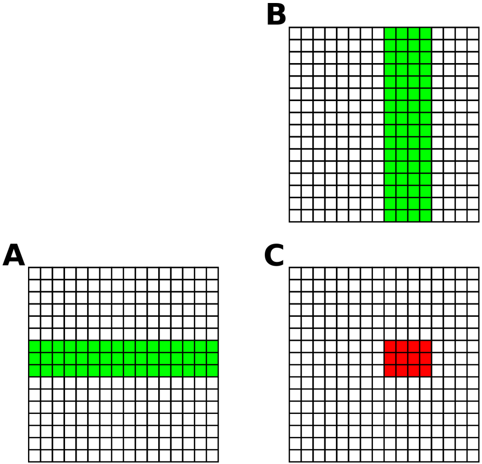
Let A, B, C be $N \times N$ matrices.



- The multiplication $C = AB$ corresponds to the computation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Second kernel: Use P threads per block, each block computing P consecutive elements of a row of C ($P = 4$ in this figure).
- Suppose that N is **not** divisible by P.

Matrix multiplication

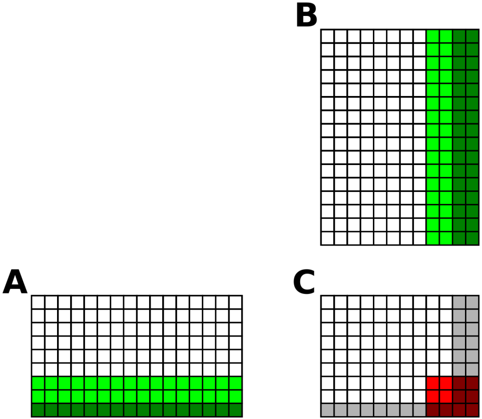
Let A, B, C be $N \times N$ matrices.



- The multiplication $C = AB$ corresponds to the computation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Fourth kernel: Use $P \times Q$ threads per block, each block computing $P \times Q$ consecutive elements in a tile of C ($P = 4$ et $Q = 3$ in this figure).
- Suppose that N is divisible by P and Q.

Matrix multiplication

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



- The multiplication $C = AB$ corresponds to the computation $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Fourth kernel: Use $P \times Q$ threads per block, each block computing $P \times Q$ consecutive elements in a tile of C ($P = 4$ et $Q = 3$ in this figure).
- Suppose that N is **not** divisible by P and Q.

Contact

Oguz Kaya
Université Paris-Saclay and LISN, Paris, France
oguz.kaya@universite-paris-saclay.fr
www.oguzkaya.com