Grilles/blocs multi-dimensionnels, coalescence

Oguz Kaya

Maître de Conférences
Université Paris-Saclay et l'Équipe ParSys du LRI, Orsay, France

université
PARIS-SACLAY

Objectifs

- Utiliser l'indexage de grilles/blocs 2 ou 3 dims
- Réaliser accès "rapide" à la mémoire globale du GPU
- Premier TP sur multiplication des matrices

université PARIS-SACLAY

Outline

université PARIS-SACLAY

Outline

Multiplication d'un tableau 1D par blocs

Récapultatif de l'exemple da multiplication d'un tableau par blocs

```
#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;
}
```

- Multiplier un tableau 1D par une grille de blocs.
- Chaque bloc multiplie 1 élément.
- Lancer le kernel avec N blocs.
- **Question:** Que ferait-on si le tableau était plutôt $A[N][N]$?

Multiplication d'un tableau 2D par blocs

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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)
{
  // 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;
}
```

- Chaque bloc multiplie 1 élément.
- Il faut lancer N^2 blocs au total.
- Chaque N blocs consécutifs multiplient une ligne de A.
- Une division et un modulus pour trouver i et j

Multiplication d'un tableau 2D par blocs 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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)
{
  // 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;
}
```

- **dim3** définit une topologie 3D d'indexage (**dim3.{x,y,z}**).
- Mettre **dim3.z = 1** pour 2D.
- Le nombre total de blocs égale à la multiplication de dimensions.
- Pas de division ni modulus pour trouver i et j

Multiplication d'un tableau 2D par blocs 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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;
}
```

- Chaque bloc multiplie blockSize éléments.
- Chaque thread multiplie 1 élément.
- Threads travaillent sur éléments consécutifs dans une **ligne**.
- Il faut lancer $N^2/blockSize$ blocs au total.
- Chaque blocs multiplient une sous-ligne de A.
- Il faut vérifier le dépassement du tableau.

Multiplication d'un tableau 2D par blocs 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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;
}
```

- Chaque thread multiplie 1 élément.
- Il faut lancer $N^2/blockSize$ blocs au total.
- Threads travaillent sur éléments consécutifs dans une **colonne**.
- Chaque blocs multiplient une sous-colonne de A.
- Lequel est mieux (sous-ligne vs. sous-colonne)?
- Qu'est-ce qui se passe si A a peu de lignes/colonnes?

Multiplication d'un tableau 2D par blocs 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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)
{
  // 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;
}
```

- Répartir 1024 threads en 2D.
- Chaque block s'occupe d'une sous-matrice 32×32.
- Threads consécutifs travaille sur une **sous-ligne**.
- Trover i et j nécessite division et modulus.

Multiplication d'un tableau 2D par blocs 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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)
{
  // 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;
}
```

- Répartir 1024 threads en 2D.
- Chaque block s'occupe d'une sous-matrice 32×32.
- Threads consécutifs travaille sur une **sous-colonne**.
- Trover i et j nécessite division et modulus.
- Lequel est mieux (accès sous-ligne vs. sous-colonne)?

Multiplication d'un tableau 2D par blocs 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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);
  // Rapatrier le tableau multiplié vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Utiliser un **dim3** pour l'indexage 2D de threads.
- Threads consécutifs en **threadIdx.x** sont exécutés dans un **warp** (puis en **threadIdx.y**, puis en **threadIdx.z**).
- Donc chaque warp accède à une **sous-colonne** de A.

Multiplication d'un tableau 2D par blocs 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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);
  // Rapatrier le tableau multiplié vers le CPU
  cudaMemcpyFromSymbol(A, dA, N * N * sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  printf("%f\n", A[1][2]);
  return 0;
}
```

- Utiliser un **dim3** pour l'indexage 2D de threads.
- Threads consécutifs en **threadIdx.x** sont exécutés dans un **warp** (puis en **threadIdx.y**, puis en **threadIdx.z**).
- Donc chaque warp accède à une **sous-ligne** de A.
- Lequel est mieux?

Limites de dimensions de grille et de bloc

Pour une grille, il faut utiliser

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

Pour un bloc, il faut utiliser

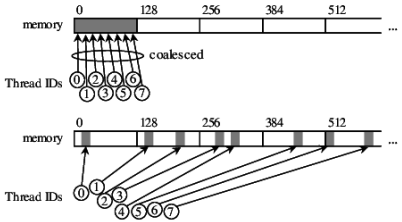
- **dim3.x** ≤ 1024
- **dim3.y** ≤ 1024
- **dim3.z** ≤ 64
- Nombre total de threads ≤ 1024

université PARIS-SACLAY

Outline

Coalescence

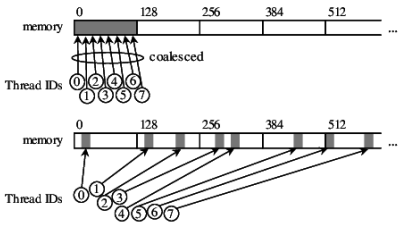
Il s'agit d'accès à la mémoire globale des threads dans un warp.



- Threads dans un warp exécute les instructions de manière **synchrone**.
- Chaque accès mémoire est également traité de manière **synchrone**.
- Si les threads accèdent à des éléments consécutifs dans la mémoire, ceci nécessite la lécture d'une ligne (donc 1 accès).

Coalescence (cont.)

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



- S'il y a des sauts, chaque ligne touchée sera lue.
  - À la limite, 32 lignes lues pour un accès.
  - La plupart d'éléments dans la ligne sont inutilisés.
- **Règle:** Concevoir le kernel tel que les accès sont contigus en **threadIdx.x**.

Exemple: Multiplication d'un tableau 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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;
}
```

- Est-il **coalescent**?
- Oui! La matrice est stockée par lignes, **threadIdx.x** aligné sur lignes.

Exemple: Multiplication d'un tableau 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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;
}
```

- Est-il **coalescent**?
- Non! La matrice est stockée par lignes, **threadIdx.x** aligné sur colonnes.
- 32 léctures mémoires nécessaires pour chaque accès mémoire d'un warp.

Exemple: Multiplication d'un tableau 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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]);
```

- Est-il **coalescent**?
- Non! La matrice est stockée par lignes, **threadIdx.x** aligné sur colonnes.
- 32 léctures mémoires nécessaires pour chaque accès mémoire d'un warp.

Exemple: Multiplication d'un tableau 2D

Multiplier chaque élément d'un tableau $A[N][N]$ par un scalaire 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]);
```

- Est-il **coalescent**?
- Oui! La matrice est stockée par lignes, **threadIdx.x** aligné sur lignes.

Exemple: Accès à un tableau avec sauts

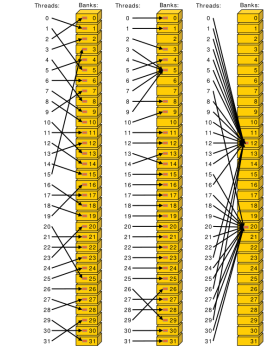
```
__device__ float dA[N];

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

- Comment s'évolue la performance en fonction de **stride**?
- Pour **stride** = 1, lécture d'une ligne de 128 octets.
- Pour **stride** = 2, lécture de deux lignes de 128 octets (moitié inutilisé)
- ...
- Pour **stride** = 32, lécture de 32 lignes de 128 octets (31/32 inutilisé)

Performance de l'accès à un tableau avec sauts

- La bande passante effective chute très rapidement avec sauts.

Règles de coalescence

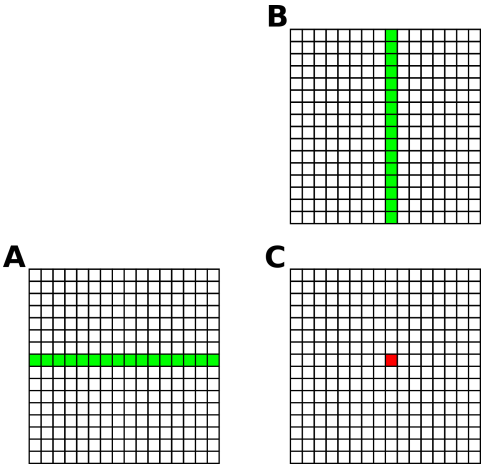


- Les threads dans un warp accèdent à la **même case mémoire** = bonnes performances (pourtant bande passent toujours gaspillée).
- Les threads dans un warp accèdent à la **même ligne en mémoire** dans un ordre **aléatoire** = toujours bonnes performances dans les nouvelles architectures.
- Si coalescence est difficile de faire, **shared memory** pourrait être utile (à venir).

Outline

Multiplication des matrices

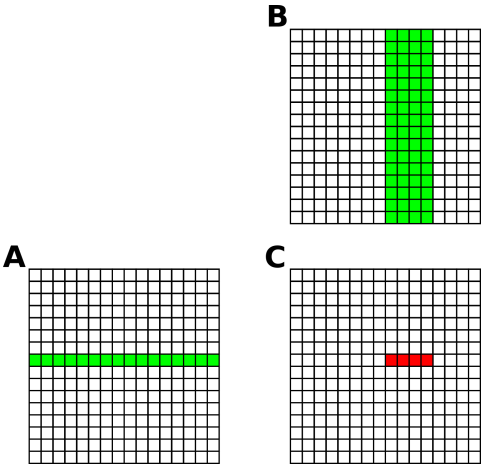
Soit A, B, C matrices de taille $N \times N$.



- La multiplication $C = AB$ corresponde au calcul $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Premier kernel: Créer un bloc pour chaque $C[i][j]$.

Multiplication des matrices

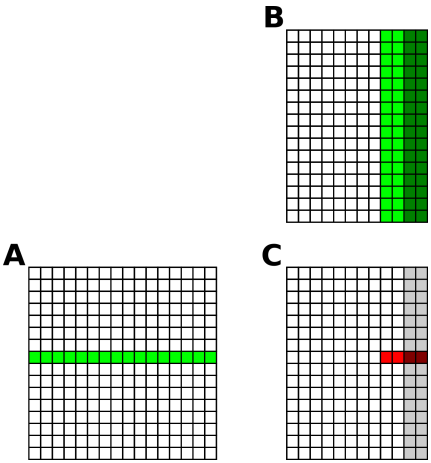
Soit A, B, C matrices de taille $N \times N$.



- La multiplication $C = AB$ corresponde au calcul $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Deuxième kernel: Utiliser P threads par bloc, chaque bloc calcul P éléments consécutifs d'une ligne de C ($P = 4$ ici).
- Supposer que P divise N.

Multiplication des matrices

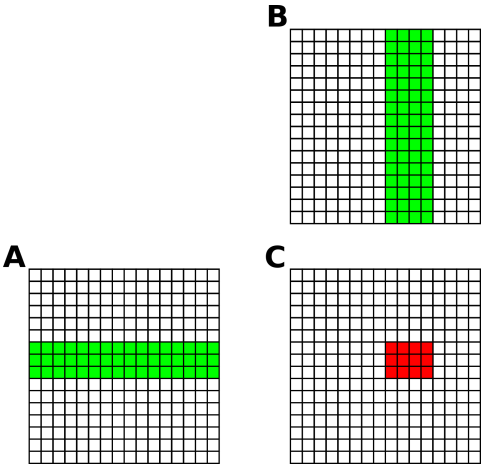
Soit A, B, C matrices de taille $N \times N$.



- La multiplication $C = AB$ corresponde au calcul $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Troisième kernel: Utiliser P threads par bloc, chaque bloc calcul P éléments consécutifs d'une ligne de C ($P = 4$ ici).
- Supposer que P ne divise pas N.

Multiplication des matrices

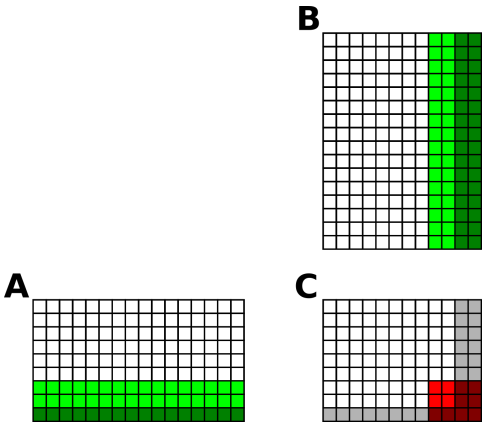
Soit A, B, C matrices de taille $N \times N$.



- La multiplication $C = AB$ corresponde au calcul $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Quatrième kernel: Utiliser PQ threads par bloc, chaque bloc calcul $P \times Q$ éléments consécutifs d'un sous-bloc de C ($P = 4$ et $Q = 3$ ici).
- Supposer que P et Q divisent N.

Multiplication des matrices

Soit A, B, C matrices de taille $N \times N$.



- La multiplication $C = AB$ corresponde au calcul $C[i][j] = \sum_{k=0}^{N-1} A[i][k]B[k][j]$.
- Cinqième kernel: Utiliser PQ threads par bloc, chaque bloc calcul $P \times Q$ éléments consécutifs d'un sous-bloc de C ($P = 4$ et $Q = 3$ ici).
- Supposer que ni P ni Q divise N.

Contact

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