Master Informatique des Systèmes Embarqués

Programmation d’architecture SIMD

CUDA

Farès Belhadj

Cours et TPs

29 octobre 2019

Plan

Présentation

Les GPUs (NVidia)

Le modèle de programmation CUDA

Démarrage avec CUDA

Farès Belhadj

Présentation

Le GPGPU en général

Farès Belhadj

GPGPU 1/5

General-Purpose Computing on Graphics Processing Units

Calculs (au sens large) sur processeurs graphiques (*Stream processing*)

Les débuts (*OpenGL*, `https://www.khronos.org/registry/OpenGL/index_gl.php`) :

- **Avant OpenGL 2** apparition en `GL_NV` (`GL_NV_vertex_program` – 2000), `GL_EXT` (*extensions*) puis `GL_ARB` (*Architecture Review Board*) des shaders (aussi NVidia *Cg* pour *C for graphics* – 2003) ;
- **OpenGL 2 (2004)** le *GLSL* ;
- **OpenGL 3.2/3.3 (2009-2010)** l'*API Core Profile* et l'*API Compatibility Profile* ;
- **OpenGL 4.3 (2013)** le *Compute Shader* (slides de cours) ;

GPGPU 2/5

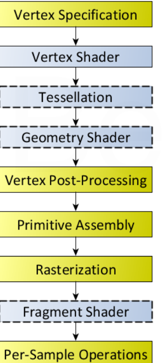
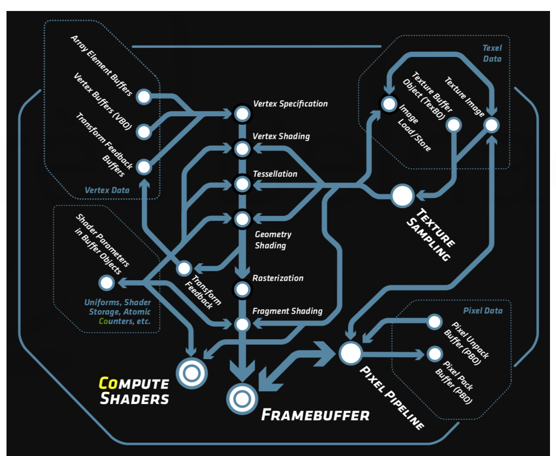


Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages.

(https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview)

Farès Belhadj

GPGPU 3/5



OpenGL 4.3 API Specifications.

(https://www.khronos.org/registry/OpenGL/specs/gl/glspec43.core.pdf)

GPGPU 4/5

Close-To-Metal (2006) puis *AMD Stream* est une API bas-niveau permettant, en contournant les drivers graphiques, de programmer directement les architectures ATI pour faire du calcul parallèle sur du flux de données ;

CUDA (2007) plateforme de développement permettant de compiler (initialement du C, voir OpenACC) et exécuter en SIMD des programmes écrits pour une architecture GPU unifiée NVidia ;

Farès Belhadj

GPGPU 5/5

OpenCL (2008) framework permettant de dispatcher des calculs (SIMD ou MIMD) sur un ensemble hétérogène de composants : CPUs, GPUs, DSPs, FPGAs, ... (voir Khronos/**Clang**) ;

Vulkan (2015) ou *OpenGL next* est une API graphique et de calcul permettant, par rapport à *OpenGL*, un contrôle plus fin sur le pipeline de traitement de données et sur l'ordonnancement CPU/GPU (voir aussi *Metal* et *Mantle*) ;

Farès Belhadj

Vulkan 1.0

Vulkan Explicit GPU Control

	OpenGL / OpenGL ES	Vulkan	
Complex drivers lead to driver overhead and cross vendor unpredictability	Application	Application responsible for memory allocation and thread management to generate command buffers	Simpler drivers for low-overhead efficiency and cross vendor portability
Error management is always active	Traditional graphics drivers include significant context, memory and error management		Layered architecture so validation and debug layers can be unloaded when not needed
Driver processes full shading language source		Direct GPU Control	Run-time only has to ingest SPIR-V intermediate language
Separate APIs for desktop and mobile markets	GPU	GPU	Unified API for mobile, desktop, console and embedded platforms

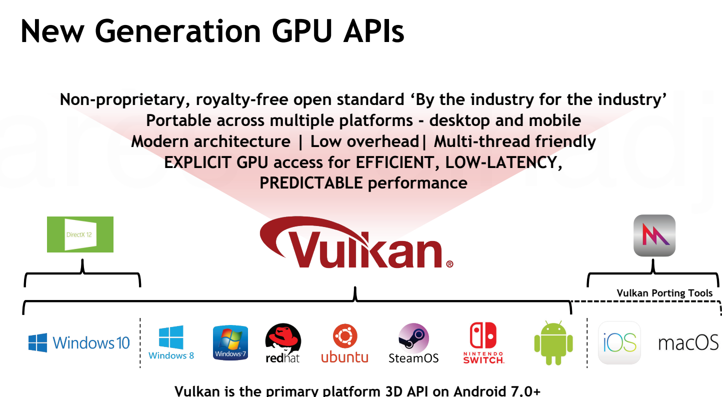
Vulkan delivers the maximized performance and cross platform portability needed by sophisticated engines, middleware and apps

KHRONOS GROUP

© Copyright Khronos Group 2015 - Page 3

https://www.khronos.org/assets/uploads/developers/library/overview/2015_vulkan_v1_Overview.pdf

Vulkan 1.1 (1/2)



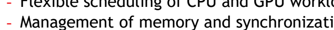
https://www.khronos.org/assets/uploads/developers/library/overview/Vulkan-1-1-Presentation_Mar18.pdf

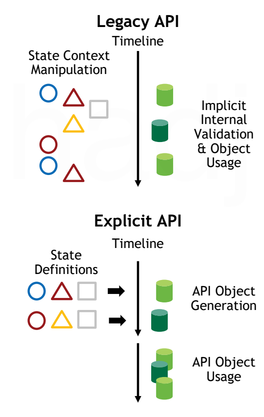
Vulkan 1.1 (2/2)

KHRONOS GROUP

Explicit GPU Access

- **Application tells the driver what it is going to do**
  - In enough detail that driver doesn't have to guess
  - When the driver needs to know it
- **In return, driver promises to do**
  - What the application asks for
  - When it asks for it
  - Very quickly
- **Predictable performance costs**
  - Creating pipelines, allocating memory, …
- **No driver magic - no surprises - simpler drivers**
  - Remove guesswork and late decision-making
- **Putting control in the hands of developers**
  - Flexible scheduling of CPU and GPU workloads
  - Management of memory and synchronization





© Copyright Khronos™ Group 2018 - Page 4

https://www.khronos.org/assets/uploads/developers/library/overview/Vulkan-1-1-Presentation_Mar18.pdf

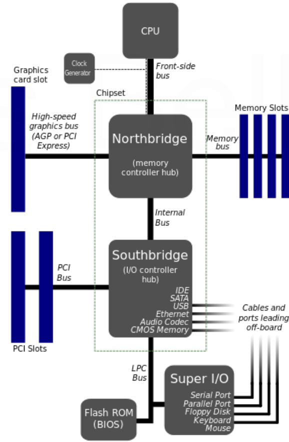
Les GPUs (NVidia)

Les GPU en général, les GPUs NVidia en particulier

Zoom sur l’architecture Kepler

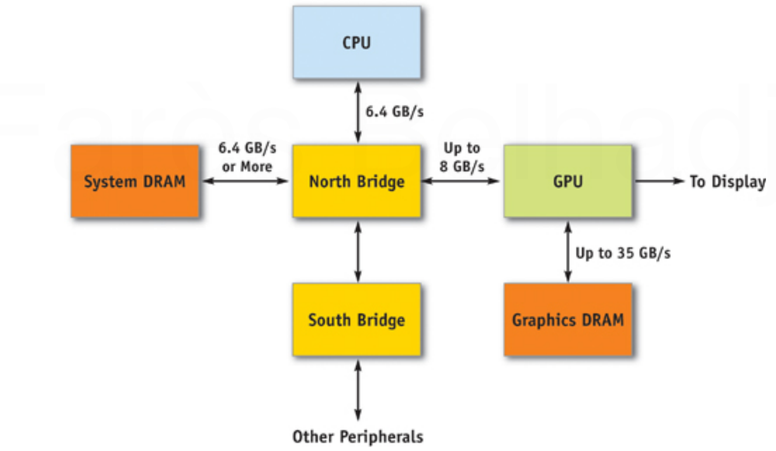
Farès Belhadj

La place du GPU au sein d'une architecture Northbridge/Southbridge



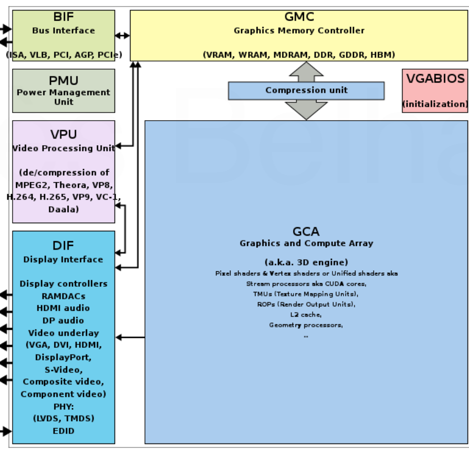
https://en.wikipedia.org/wiki/Graphics_processing_unit

Architecture d'une carte mère avec GPU dédié (2005)



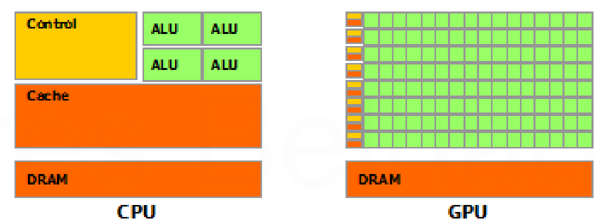
GPU Gems 2 (auteur Matt Pharr, éditeur Addison Wesley)

Composants d'un GPU



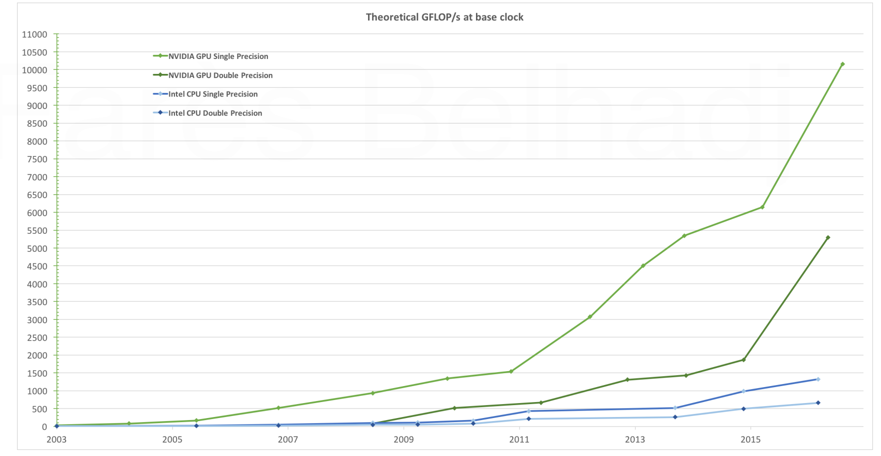
https://en.wikipedia.org/wiki/Graphics_processing_unit

Transistors pour le traitement de données



https://docs.nvidia.com/cuda/cuda-c-programming-guide

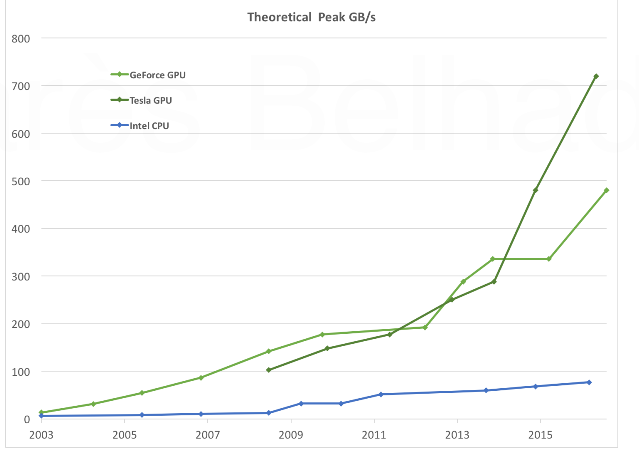
Opérations par seconde sur les flottants



https://docs.nvidia.com/cuda/cuda-c-programming-guide

Mémoire : bande passante

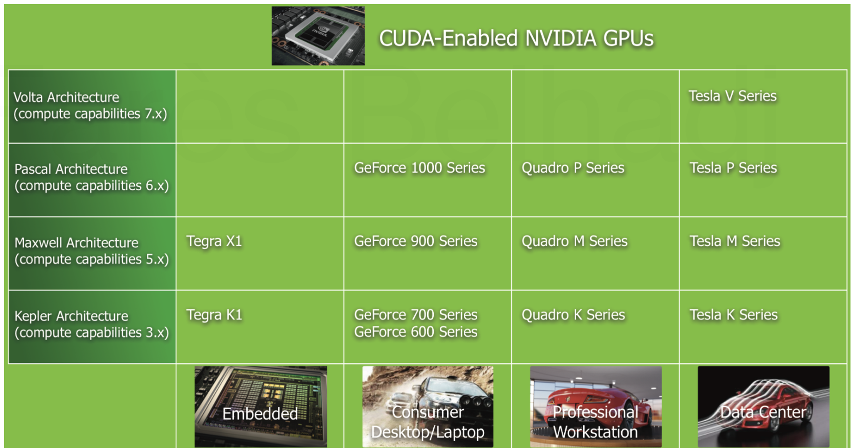
(rapport entre mémoire partagée et dédiée : approx. $x10$)



https://docs.nvidia.com/cuda/cuda-c-programming-guide

Farès Belhadj

Les GPU NVidia CUDA-Ready 2018



https://docs.nvidia.com/cuda/cuda-c-programming-guide

Les GPU NVidia CUDA-Ready 2019

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)	
CUDA-enabled NVIDIA GPUs						

	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series

https://docs.nvidia.com/cuda/cuda-c-programming-guide

Compute Capabilities (1/2)

Table 13. Feature Support per Compute Capability

<table>
<tr><th>Feature Support</th><th colspan="6">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>3.0</th><th>3.2</th><th>3.5, 3.7, 5.0, 5.2</th><th>5.3</th><th>6.x</th><th>7.x</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td colspan="6">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td><td colspan="6">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td><td colspan="6">Yes</td></tr>
<tr><td>Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())</td><td colspan="4">No</td><td colspan="2">Yes</td></tr>
<tr><td>Warp vote and ballot functions (Warp Vote Functions)</td><td colspan="6" rowspan="6">Yes</td></tr>
<tr><td>__threadfence_system() (Memory Fence Functions)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Synchronization Functions)</td></tr>
<tr><td>Surface functions (Surface Functions)</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
<tr><td>Unified Memory Programming</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td>No</td><td colspan="5">Yes</td></tr>
<tr><td>Dynamic Parallelism</td><td colspan="2">No</td><td colspan="4">Yes</td></tr>
<tr><td>Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td><td colspan="3">No</td><td colspan="3">Yes</td></tr>
<tr><td>Tensor Core</td><td colspan="5">No</td><td>Yes</td></tr>
</table>

https://docs.nvidia.com/cuda/cuda-c-programming-guide

Compute Capabilities (2/2)

Table 14. Technical Specifications per Compute Capability

Technical Specifications	Compute Capability											
	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128	
Maximum dimensionality of grid of thread blocks	3											
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$											
Maximum y- or z-dimension of a grid of thread blocks	65535											
Maximum dimensionality of thread block	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block [27]	48 KB										96 KB	64 KB
Number of shared memory banks	32											
Amount of local memory per thread	512 KB											
Constant memory size	64 KB											
Cache working set per multiprocessor for constant memory	8 KB							4 KB	8 KB			
Cache working set per multiprocessor for texture memory	Between 12 KB and 48 KB							Between 24 KB and 48 KB			32 - 128 KB	32 or 64 KB
Maximum width for a 1D texture reference bound to a CUDA array	65536											
Maximum width for a 1D texture reference bound to linear memory	2^{27}											
Maximum width and number of layers for a 1D layered texture reference	16384 x 2048											
Maximum width and height for a 2D texture reference bound to a CUDA array	65536 x 65535											
Maximum width and height for a 2D texture reference bound to linear memory	65000 x 65000											
Maximum width and height for a 2D texture reference bound to a CUDA array supporting texture gather	16384 x 16384											
Maximum width, height, and number of layers for a 2D layered texture reference	16384 x 16384 x 2048											
Maximum width, height, and depth for a 3D texture reference bound to a CUDA array	4096 x 4096 x 4096											
Maximum width (and height) for a cubemap texture reference	16384											
Maximum width (and height) and number of layers for a cubemap layered texture reference	16384 x 2046											
Maximum number of textures that can be bound to a kernel	256											
Maximum width for a 1D surface reference bound to a CUDA array	65536											
Maximum width and number of layers for a 1D layered surface reference	65536 x 2048											
Maximum width and height for a 2D surface reference bound to a CUDA array	65536 x 32768											
Maximum width, height, and number of layers for a 2D layered surface reference	65536 x 32768 x 2048											
Maximum width, height, and depth for a 3D surface reference bound to a CUDA array	65536 x 32768 x 2048											
Maximum width (and height) for a cubemap surface reference bound to a CUDA array	32768											
Maximum width (and height) and number of layers for a cubemap layered surface reference	32768 x 2046											
Maximum number of surfaces that can be bound to a kernel	16											
Maximum number of instructions per kernel	512 million											

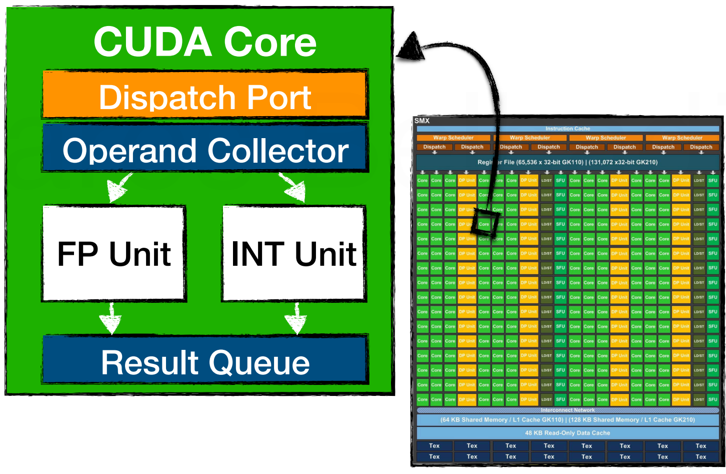
https://docs.nvidia.com/cuda/cuda-c-programming-guide

Zoom sur une architecture Kepler

SMX (où SM = *Streaming Multiprocessor*) : 192 cores CUDA simple précision, 64 unités double précision, 32 SFUs (unités pour fonctions spécifiques), 32 unités de chargement/stockage. Support total de la norme IEEE 754-2008.

https://images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

Zoom sur SMX et CUDA Core



https://images.nvidia.com/content/pdf/tesla/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

Les warps

- *Warp* : ensemble de 32 *threads* exécutables en mode SIMD
- Sur l'architecture Kepler, 4 *Warp schedulers* permettent à 4 *warps* d'être exécutés simultanément
- Deux *Dispatchers* par *Warp scheduler* permettent l'envoi de deux instructions indépendantes par cycle
- Kepler gère les instructions à double précision et permet de les associer à d'autres instructions
- *cf.* le nombre de *warps* par SMX selon les *Compute Capabilities*

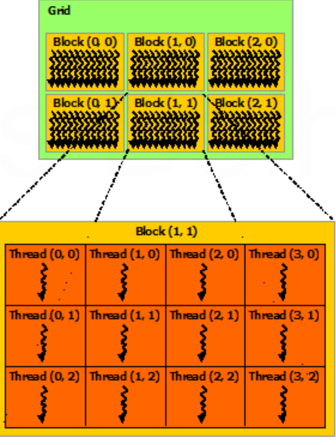
Fares Belhadj

Le modèle de programmation CUDA

Compute ***Unified*** *Device Architecture*

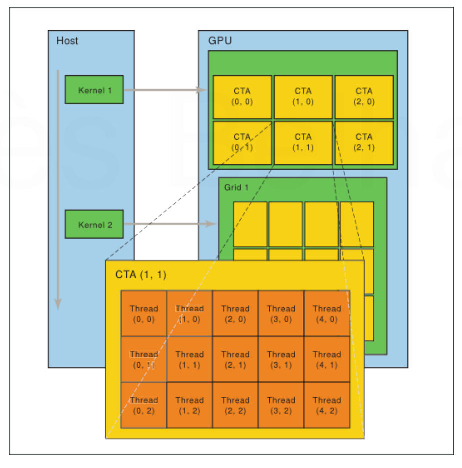
Farès Belhadj

Les Kernels : hiérarchie des Threads



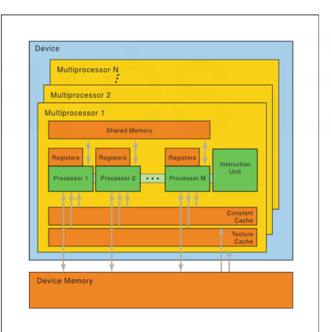
Farès Belhadj

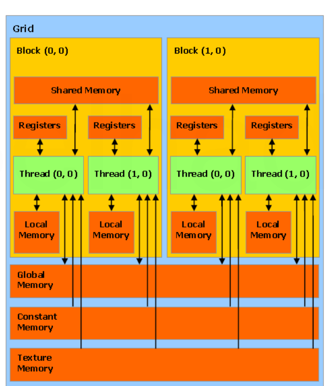
Organisation Host / Device



Farès Belhadj

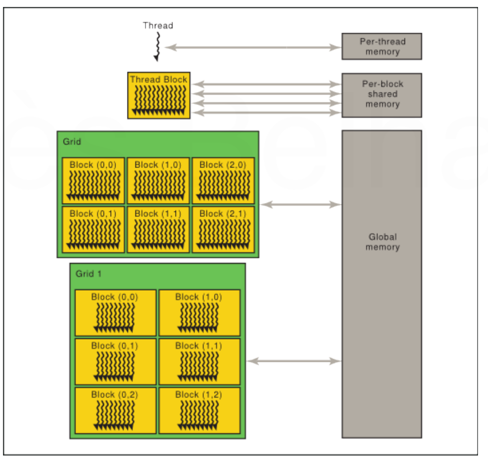
La mémoire





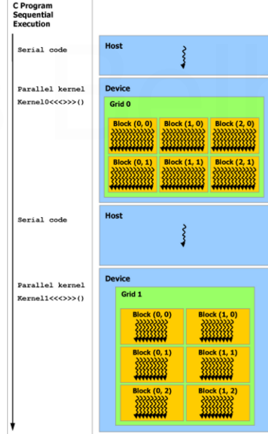
⚠ *Local memory* : portion de mémoire globale propre au *thread*

Utilisation de la mémoire



Farès Belhadj

Modèle hétérogène de programmation : exécution de *kernels*



Farès Belhadj

Résumé

Optimiser un programme CUDA revient (aussi) à :

- repenser le problème autrement, le découper, l'organiser en blocs ;
- bien choisir et optimiser les accès à la mémoire ;
- avoir une réflexion sur le nombre de blocs et leurs tailles (latence vs disponibilité des registres) ;
- bien disposer les *threads* pour optimiser l'utilisation du cache (voisinage) ;
- coder et acquérir les bonnes pratiques avec l'expérience.

Farès Belhadj

Démarrage avec CUDA

Installation, quelques mots clés, premiers exemples

Farès Belhadj

Où se trouve CUDA ? (compilateur, runtime, driver)

- liens pour installer CUDA sur votre machine
  (Attention à *High Sierra* ou *Mojave*)
  `https://developer.nvidia.com/how-to-cuda-c-cpp`

  * `https://docs.nvidia.com/cuda/index.html#installation-guides`
- CUDA en **A193 ou A160** : dans `/usr/local/cuda`, configurez vos variables d'environnement et copiez le dossier `Samples`
- La machine `cuda` : le serveur de calcul `XXXXXXXXXXX` est accessible depuis le réseau de la formation ou en faisant un rebond sur `calcium`. Il possède deux cartes Tesla K80. L'installation de CUDA se trouve dans `/usr/local/cuda`. Ajoutez ces lignes à votre `.bash_profile` :

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

Fonctions : spécification de l'espace d'exécution (1/2)

`__global__` ce *specifier* déclare une fonction comme un *kernel*.

- Cette fonction est appelée depuis le *host* (le CPU) et est exécutée sur le *device* (le GPU).
- Elle peut être appelée depuis le *device* sur une architecture ayant un *compute capability* de 3.2 et plus (*cf. Dynamic Parallelism*).
- Elle doit être de type `void` et ne être membre d'une classe.
- Un appel à cette fonction doit spécifier une configuration d'exécution (nombre de blocs et de *threads* par bloc).
- Un appel à cette fonction est asynchrone (l'appel se termine avant la fin de l'exécution dans le *device*).

`__device__` ce *specifier* permet de déclarer une fonction exécutée sur le *device* (le GPU). Cette fonction ne peut être appelée que depuis le *device* ;

les *specifiers* `__global__` et `__device__` ne peuvent être utilisés ensemble.

Fonctions : spécification de l'espace d'exécution (2/2)

`__host__` ce *specifier* déclare une fonction qui est exécutée sur le *host*.

- Cette fonction n'est appelée que depuis le *host*.
- Ce *specifier* est optionnel. Sa présence ou l'absence des autres *specifiers* indiquent que la fonction sera compilée pour le *host*.

les *specifiers* `__global__` et `__host__` ne peuvent être utilisés ensemble.

si les *specifiers* `__device__` et `__host__` sont utilisés ensemble, la fonction sera respectivement compilée pour le *device* et le *host*. La macro `__CUDA_ARCH__` permettra de différencier les codes selon le cas (voir l'exemple donné page 37).

Farès Belhadj

Utilisation de __CUDA_ARCH__ pour la différenciation des contextes d'exécution

```
__host__ __device__ func() {
#if __CUDA_ARCH__ >= 600
   // Device code path for compute capability 6.x
#elif __CUDA_ARCH__ >= 500
   // Device code path for compute capability 5.x
#elif __CUDA_ARCH__ >= 300
   // Device code path for compute capability 3.x
#elif __CUDA_ARCH__ >= 200
   // Device code path for compute capability 2.x
#elif !defined(__CUDA_ARCH__)
   // Host code path
#endif
}
```

Fares Belhadj

Le kernel

Pour une fonction *kernel* telle que :

```
__global__ void foo(void) {
}
```

un appel à cette fonction se fait avec :

```
foo <<< nbBlocs, nbThreadsParBloc >>> ();
```

Les valeurs comprises entre $<<<$ et $>>>$ indiquent respectivement le nombre de blocs et le nombre de *threads* par bloc à utiliser lors l'exécution. Pour rappel, les blocs ne peuvent pas communiquer entre-eux. Enfin, pendant l'exécution de `foo` dans un *thread*, nous aurons accès à trois variables d'environnement `blockIdx`, `threadIdx` et `blockDim` donnant respectivement l'indice du bloc courant, l'indice du *thread* courant et le nombre de *threads* par bloc.

Farès Belhadj

Mon premier programme CUDA : hello.cu

```
#include <stdio.h>
__global__ void foo(void) {
}
int main(void) {
  foo <<<1, 1>>>();
  printf("Hello CUDA\n");
  return 0;
}
```

Sa compilation et exécution sur la machine `cuda` :

```
-bash-4.1$ nvcc hello.cu && ./a.out
Hello CUDA
-bash-4.1$
```

Farès Belhadj

Addition de deux valeurs dans le GPU

```
__global__ void add(float * pa, float * pb, float * pc) {
  *pc = *pa + *pb;
}
int main(void) {
  float a = 1, b = 2, c;
  add <<<1, 1>>>(&a, &b, &c);
  return 0;
}
```

Farès Belhadj

Addition de deux valeurs, un doute ? Modifions le main

```
...
#include <assert.h>
int main(void) {
  float a = 1, b = 2, c;
  add <<<1, 1>>>(&a, &b, &c);
  assert(c == a + b);
  return 0;
}
```

Alors???

Farès Belhadj

Revenons à la mémoire

Les “contextes mémoire” sont différents selon qu’on soit sur le CPU (le *host*) ou le GPU (le *device*).

- `pa` contient bien l’adresse de `a` (`pb` celle de `b` ...), mais cette adresse ne correspond pas à `a` dans le contexte du *device*.
- Les pointeurs dans le contexte *device* pointent sur la mémoire GPU.
- Les pointeurs dans le contexte *host* pointent sur la mémoire CPU.
- Il est nécessaire de mettre en place des transferts entre mémoire CPU et mémoire GPU.

Fares Belhadj

Deuxième essai

```
__device__ float dc[1];
__global__ void add(float a, float b) {
  *dc = a + b;
}
#include <assert.h>
#include <stdio.h>
int main(void) {
  float a = 1, b = 2, c,;
  add <<<1, 1>>>(a, b);
  cudaMemcpyFromSymbol(&c, dc, sizeof c);
  assert(c == a + b);
  return 0;
}
```

Farès Belhadj

Nouveau : utilisation de `__device__` pour une variable; utilisation de la fonction `cudaMemcpyFromSymbol`.

API Reference Manual

`https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf`

Voir :

- `cudaMemcpyFromSymbol`
- `cudaMemcpyToSymbol`
- `cudaMemcpyFromArray`
- `cudaMemcpyToArray`
- `cudaMemcpy`
- `cudaMalloc`
- `cudaFree`
- ...

Fares Belhadj

Variable Memory Space Specifiers (1/2)

plus de détails ici

`__device__` variable résidant dans le *device*. Ce *specifier* peut être combiné avec l'un des suivants afin de déterminer le type de mémoire dans lequel réside la variable. Sinon :

- Resides in global memory space ;
- Has the lifetime of the CUDA context in which it is created ;
- Has a distinct object per device ;
- Is accessible from all the threads within the grid and from the host through the runtime library `cudaGetSymbolAddress`, `cudaGetSymbolSize`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`.

`__constant__` variable qui :

- Resides in constant memory space ;
- Idem que `__device__` pour le reste.

Variable Memory Space Specifiers (2/2)

`__shared__` variable qui :

- Resides in the shared memory space of a thread block,
- Has the lifetime of the block,
- Has a distinct object per block,
- Is only accessible from all the threads within the block,
- Does not have a constant address, donc n'est pas accessible depuis le *host*.

`__managed__` variable qui :

- Can be referenced from both device and host code, e.g., its address can be taken or it can be read or written directly from a device or host function.
- Has the lifetime of an application.

`__restrict__` *cf.* problème d'*aliasing* cité ici.

Farès Belhadj

Alternative au deuxième essai

```
__managed__ float mc;
__global__ void add(float a, float b) {
  mc = a + b;
}
#include <assert.h>
int main(void) {
  float a = 1, b = 2;
  add <<<1, 1>>>(a, b);
  cudaDeviceSynchronize();
  assert(mc == a + b);
  return 0;
}
```

Nouveau : utilisation du *specifier* de variable `__managed__` ; utilisation de la fonction `cudaDeviceSynchronize`. Pour compiler, ajoutez `-arch=sm_30` à la ligne de compilation.

Farès Belhadj

Une autre alternative

```
__device__ __managed__ float a = 1, b = 2, c;
__global__ void add(float *a, float *b, float *c) {
  *c = *a + *b;
}
#include <assert.h>
int main(void) {
  add <<<1, 1>>>(&a, &b, &c);
  cudaDeviceSynchronize();
  assert(c == a + b);
  return 0;
}
```

Remarque : nous ne pouvons pas utiliser le *specifier* de variable `__managed__` à l'intérieur d'un bloc de fonction.

Farès Belhadj

Types de vecteurs et variables intégrées

plus de détails ici pour les type, et là pour les *Built-in variables*.

Farès Belhadj

Exercice 1 (inspiré des exemples NVidia)

Écrire un programme calculant la somme de deux vecteurs de dimension N dans un troisième vecteur.

Farès Belhadj

Addition de 2 vecteurs de dimension N : `vecAdd1.cu`

```
__global__ void vecAdd(float * A, float * B, float * C) {
  int i = blockIdx.x;
  C[i] = A[i] + B[i];
}
#define N 1024
#include <stdlib.h>
#include <assert.h>
int main(void) {
  int i;
  float A[N], B[N], C[N], *devA, *devB, *devC;
  for(i = 0; i < N; ++i) {
    A[i] = rand(); B[i] = rand();
  }
  cudaMalloc(&devA, N * sizeof *devA);
  cudaMalloc(&devB, N * sizeof *devB);
  cudaMalloc(&devC, N * sizeof *devC);
  cudaMemcpy(devA, A, N * sizeof *devA, cudaMemcpyHostToDevice);
  cudaMemcpy(devB, B, N * sizeof *devB, cudaMemcpyHostToDevice);
  vecAdd <<<N, 1>>>(devA, devB, devC);
  cudaMemcpy(C, devC, N * sizeof *devC, cudaMemcpyDeviceToHost);
  /* pourquoi un cudaDeviceSynchronize() n'est pas necessaire ? */
  cudaFree(devA); cudaFree(devB); cudaFree(devC);
  for(i = 0; i < N; ++i)
    assert(C[i] == (A[i] + B[i]));
  return 0;
}
```

Farès Belhadj

Rendons dynamique le choix du nombre de blocs et threads : `vecAdd2.cu`

```
...
  int i = blockIdx.x * blockDim.x + threadIdx.x;
...
int main(int argc, char ** argv) {
  int i, N, Th;
  float *A, *B, *C, *devA, *devB, *devC;
  if(argc != 3 ||
     ( (N = atoi(argv[1])) < 9 || N > 27 ) ||
     ( (Th = atoi(argv[2])) < 0 || Th > 10 ) ) {
    fprintf(stderr, "usage: %s <N> <Th>\navec 8 < N < 28 et 0 <= Th <= 10\n", argv[0]);
    return 1;
  }
...
  N  = 1 << N;
  Th = 1 << Th;
  A = (float *)malloc(N * sizeof *A);
...
  vecAdd <<<N / Th, Th>>>(devA, devB, devC);
...
```

Farès Belhadj

Ajoutons le temps ... : vecAdd2.cu

```
#include <sys/time.h>
static void   initTime(void);
static double getTime(void);
static struct timeval ti;
void initTime(void) {
  gettimeofday(&ti, (struct timezone*) 0);
}
double getTime(void) {
  struct timeval t;
  double diff;
  gettimeofday(&t, (struct timezone*) 0);
  diff = (t.tv_sec - ti.tv_sec) * 1000000
    + (t.tv_usec - ti.tv_usec);
  return diff/1000.;
}
...
  for(i = 0; i < N; ++i) {
    A[i] = rand(); B[i] = rand();
  }
  initTime();
...
  cudaFree(devA); cudaFree(devB); cudaFree(devC);
  printf("%d\t%F\n", N, getTime());
```

Farès Belhadj

Et si on l'utilisait comme ça ?

```
#!/bin/sh
rm -f thTest*
nvcc vecAdd2.cu

for((th=0;th<11;++th))
do
    for((i=10;i<28;++i))
    do
        ./a.out $i $th >> thTest$th
    done
done
```

Farès Belhadj

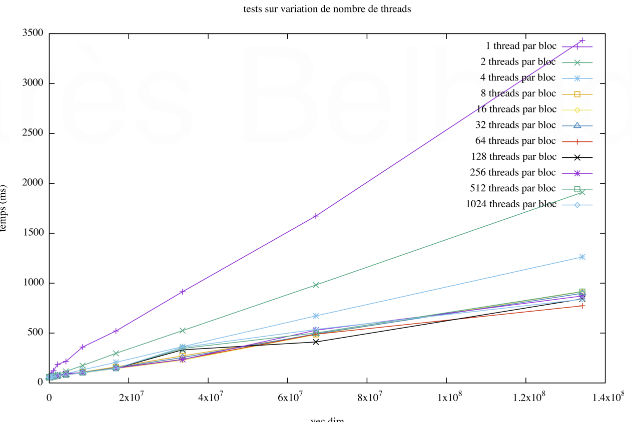
GNUPlot pour rendre les courbes de temps : `thTest.plot`

Farès Belhadj

```
set xlabel "vec dim"
set ylabel "temps"

set autoscale

set title "tests sur variation de nombre de threads"

set style data linespoints

plot "thTest0" using 1:2 title "1 thread par bloc", \
     "thTest1" using 1:2 title "2 threads par bloc", \
     "thTest2" using 1:2 title "4 threads par bloc", \
     "thTest3" using 1:2 title "8 threads par bloc", \
     "thTest4" using 1:2 title "16 threads par bloc", \
     "thTest5" using 1:2 title "32 threads par bloc", \
     "thTest6" using 1:2 title "64 threads par bloc", \
     "thTest7" using 1:2 title "128 threads par bloc", \
     "thTest8" using 1:2 title "256 threads par bloc", \
     "thTest9" using 1:2 title "512 threads par bloc", \
     "thTest10" using 1:2 title "1024 threads par bloc"
```

$ gnuplot thTest.plot

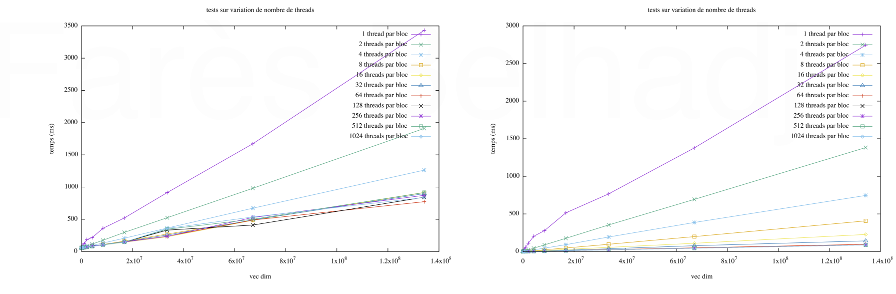
(tests sur NVIDIA GeForce GT 750M 2048 MB)



Lien vers l'ensemble des éléments de cet exemple

$ gnuplot thTest.plot

en excluant les transferts mémoire de la mesure de temps



incluant les tranferts mémoire

excluant les transferts mémoire

Lecture de code

Les *samples* fournis avec le CUDA Toolkit

1. Copiez le dossier `samples` depuis l'install du Toolkit (`/usr/local/cuda/` sous Linux ou `/Developer/NVIDIA/CUDA/` sous Mas OS X) vers quelque part chez vous ;
2. Commençons par aller voir dans `0_Simple` :
   2.1 Lecture du code `simplePrintf` ;
   2.2 Lecture du code `vectorAdd` ;
   2.3 Lecture de la curiosité `vectorAddDrv`.
3. Compilez, exécutez d'autres *samples* : `1_Utilities/deviceQuery`, `1_Utilities/bandwidthTest`, `5_Simulations/fluidsGL` ...

Farès Belhadj

Mon Laptop (1/3) - deviceQuery

```
macbook-pro-de-fares:deviceQuery amsi$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: ''GeForce GT 750M''
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2048 MBytes (2147024896 bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            926 MHz (0.93 GHz)
  Memory Clock rate:                             2508 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
```

Mon Laptop (2/3) - deviceQuery

```
  Texture alignment:                              512 bytes
  Concurrent copy and kernel execution:           Yes with 1 copy engine(s)
  Run time limit on kernels:                      Yes
  Integrated GPU sharing Host Memory:             No
  Support host page-locked memory mapping:        Yes
  Alignment requirement for Surfaces:             Yes
  Device has ECC support:                         Disabled
  Device supports Unified Addressing (UVA):       Yes
  Device supports Compute Preemption:             No
  Supports Cooperative Kernel Launch:             No
  Supports MultiDevice Co-op Kernel Launch:       No
  Device PCI Domain ID / Bus ID / location ID:    0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1, NumDevs = 1
Result = PASS
```

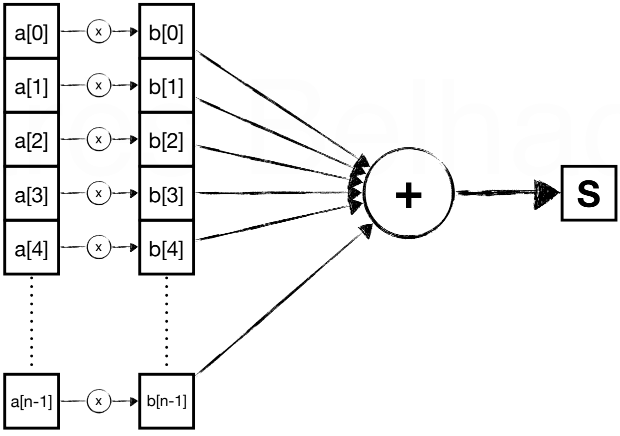
Farès Belhadj

Mon Laptop (3/3) - bandwidthTest

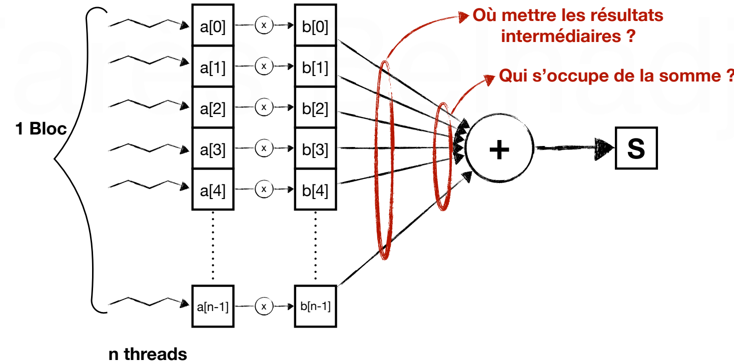
```
macbook-pro-de-fares:bandwidthTest amsi$ ./bandwidthTest
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: GeForce GT 750M
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
 Transfer Size (Bytes)Bandwidth(GB/s)
 320000003.7

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
 Transfer Size (Bytes)Bandwidth(GB/s)
 320000006.5

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
 Transfer Size (Bytes)Bandwidth(GB/s)
 3200000042.6

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enab
```

Farès Belhadj

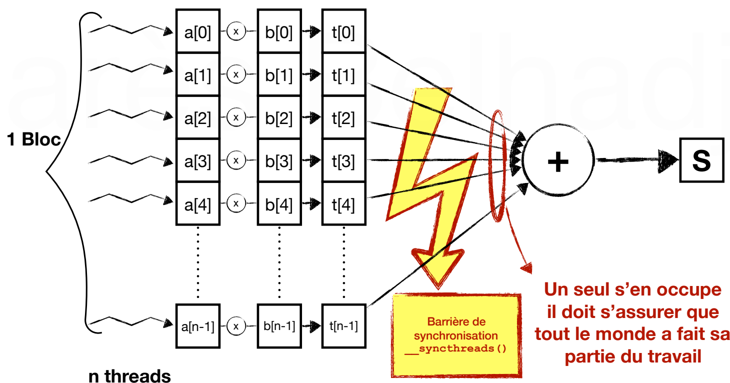
TP0 : Produit scalaire



TP0 : Produit scalaire, 1ere proposition



TP0 : Produit scalaire, essayons



TP0 : Produit scalaire, V1 (1/2)

```
__global__ void dot(float * A, float * B, float * C, float *S, int dimension) {
  int i = threadIdx.x, j;
  C[i] = A[i] * B[i];
  __syncthreads(); // barrière de synchro
  if(i == 0) { // le seul en charge de la somme
    for(j = 0, *S = 0; j < dimension; ++j)
      *S += C[j];
  }
}
int main(int argc, char ** argv) {
  const int N = 1 << 10; /* je ne peux aller que jusqu'à 1024 sur mon Laptop */
  float *A, *B, S, St, *devA, *devB, *devC, *devS;
  srand(42);
  A = (float *)malloc(N * sizeof *A); B = (float *)malloc(N * sizeof *B);
  cudaMalloc(&devA, N * sizeof *devA); cudaMalloc(&devB, N * sizeof *devB);
  cudaMalloc(&devC, N * sizeof *devC); cudaMalloc(&devS, sizeof *devS);
  for(int i = 0; i < N; ++i) {
    A[i] = rand()/(float)RAND_MAX; B[i] = rand()/(float)RAND_MAX;
  }
  cudaMemcpy(devA, A, N * sizeof *devA, cudaMemcpyHostToDevice);
  cudaMemcpy(devB, B, N * sizeof *devB, cudaMemcpyHostToDevice);
  dot <<<1, N>>>(devA, devB, devC, devS, N);
  cudaMemcpy(&S, devS, sizeof *devS, cudaMemcpyDeviceToHost);
  St = dot(A, B, N); //version __host__ de la fonction, donnée plus bas
  printf("dimension = %d, resGPU = %f, resCPU = %f\n", N, S, St);
  cudaFree(devA); cudaFree(devB); cudaFree(devC); cudaFree(devS);
  free(A); free(B);
  return 0;
}
```

Farès Belhadj

TP0 : Produit scalaire, V1 (2/2)

```
__host__ float  dot(float * A, float * B, int dimension) {
  float S = 0;
  for(int i = 0; i < dimension; ++i)
    S += A[i] * B[i];
  return S;
}
```

Les tests donnent ça :

```
MacBook-Pro-de-Fares:TP0_scalar_V1 amsi$ nvcc scalar.cu && ./a.out
dimension = 1024, resGPU = 255.132690, resCPU = 255.132690
```

En ajoutant les perfs en temps pour GPU & CPU + quelques tours :

```
MacBook-Pro-de-Fares:TP0_scalar_V1 amsi$ nvcc scalar.cu && ./a.out
dimension = 1024, resGPU = 255.132690, resCPU = 255.132690, timeGPU = 0.610000, timeCPU = 0.002000
dimension = 1024, resGPU = 253.201584, resCPU = 253.201584, timeGPU = 0.562000, timeCPU = 0.002000
dimension = 1024, resGPU = 254.537552, resCPU = 254.537552, timeGPU = 0.550000, timeCPU = 0.003000
dimension = 1024, resGPU = 255.201096, resCPU = 255.201096, timeGPU = 0.546000, timeCPU = 0.003000
dimension = 1024, resGPU = 252.147842, resCPU = 252.147842, timeGPU = 0.546000, timeCPU = 0.003000
```

Farès Belhadj

TP0 : Produit scalaire, V2 & mémoire partagée (1/2)

1. les accès à la *shared memory* est plus performant ;
2. cette mémoire est partagée par l'ensemble des thread au sein d'un bloc ;
3. la données n'est pas visible par les threads des autres blocs ;
4. ajouter le *specifier* `__shared__` (voir ci-dessous).

```
__global__ void dot(float * A, float * B, float *S, int dimension) {
  __shared__ float tmp[N]; /* N est une globale connue dans ce context */
  int i = threadIdx.x;
  tmp[i] = A[i] * B[i];
  __syncthreads();
  if(i == 0) { // le seul en charge de la somme
    *S = 0;
    for(int j = 0; j < dimension; ++j)
      *S += tmp[j];
  }
}
/* plus besoin de devC dans le reste, supprimer aussi de l'appel */
...
```

Fares Belhadj

TP0 : Produit scalaire, V2 & mémoire partagée (2/2)

Conséquence : amélioration du temps d'exécution ($\times 5$ dans cet exemple).

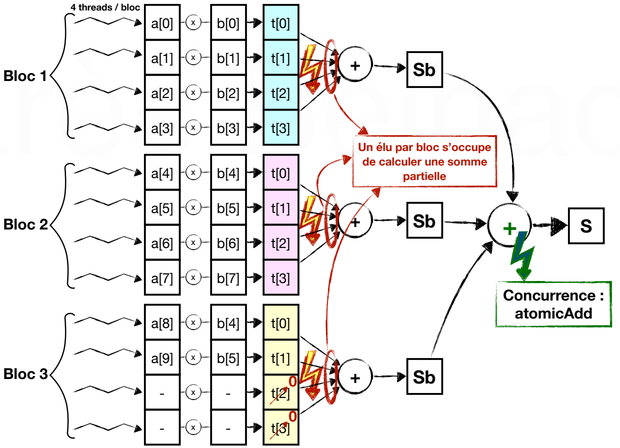
```
MacBook-Pro-de-Fares:TP0_scalar_V2 amsi$ nvcc scalar.cu && ./a.out
dimension = 1024, resGPU = 255.132690, resCPU = 255.132690, timeGPU = 0.163000, timeCPU = 0.003000
dimension = 1024, resGPU = 253.201584, resCPU = 253.201584, timeGPU = 0.116000, timeCPU = 0.003000
dimension = 1024, resGPU = 254.537552, resCPU = 254.537552, timeGPU = 0.111000, timeCPU = 0.003000
dimension = 1024, resGPU = 255.201096, resCPU = 255.201096, timeGPU = 0.137000, timeCPU = 0.003000
dimension = 1024, resGPU = 252.147842, resCPU = 252.147842, timeGPU = 0.127000, timeCPU = 0.002000
```

Rendre dynamique l'allocation de la *shared*

```
__global__ void dot(float * A, float * B, float *S, int dimension) {
   extern __shared__ float tmp[]; /* extern + pas de taille spécifiée */
   ...
}
/* plus loin lors de l'appel */
...
   dot <<<1, N, TAILLE_EN_OCTETS_DE_LA_SHARED>>>(devA, devB, devS, N);
...
```

TP0 : Produit scalaire, V3 au delà de mes 1024 (1/3)

Démultiplions les blocs, multiple espaces de définition, synchro et concurrence.



Exemple : 3 blocs, 4 threads par bloc, vecteurs (a et b) de dimension 10

TP0 : Produit scalaire, V3 au delà de mes 1024 (2/3)

Détailler : `int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;`

```
__global__ void dot(float * A, float * B, float *S, int dimension) {
  extern __shared__ float tmp[];
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i == 0) *S = 0;
  if(i < dimension)
    tmp[threadIdx.x] = A[i] * B[i];
  else
    tmp[threadIdx.x] = 0;
  __syncthreads();
  if(threadIdx.x == 0) { // un élu par bloc en charge de la somme partielle
    float sum = 0;
    for(int j = 0; j < blockDim.x; ++j)
      sum += tmp[j];
    // puis d'insérer cette somme partielle dans S via atomicAdd
    // garantissant un accès sûr en mode concurrentiel
    atomicAdd(S, sum);
  }
}
int main(int argc, char ** argv) {
  const int N = 1 << 24; /* la taille n'est plus contrainte */
  int threadsPerBlock = 256; /* les meilleurs perfs avec cette config */
  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
  ...
  dot <<<blocksPerGrid, threadsPerBlock, threadsPerBlock * sizeof *devA>>>(devA, devB, devS, N);
  ...
}
```

Farès Belhadj

TP0 : Produit scalaire, V3 au delà de mes 1024 (3/3)

Question : pourquoi les résultats sont-ils différents entre CPU et GPU ?
Performances : mieux sur mon Laptop mais moins bon que le CPU (voir plus bas). Les performances sont meilleures s'il n'y avait pas les transferts mémoire.

```
dimension = 16777216, resGPU = 4194139.500000, resCPU = 4099963.750000, timeGPU = 114.141000, timeCPU = 42
dimension = 16777216, resGPU = 4194272.500000, resCPU = 4100217.000000, timeGPU = 113.430000, timeCPU = 42
dimension = 16777216, resGPU = 4196158.000000, resCPU = 4102390.500000, timeGPU = 114.836000, timeCPU = 44
dimension = 16777216, resGPU = 4193525.250000, resCPU = 4099736.750000, timeGPU = 113.607000, timeCPU = 43
dimension = 16777216, resGPU = 4195112.000000, resCPU = 4101405.750000, timeGPU = 62.074000, timeCPU = 44.
```

Performances 2 : ci-après les performances mesurées sur le serveur `cuda` **(4 × K80).**

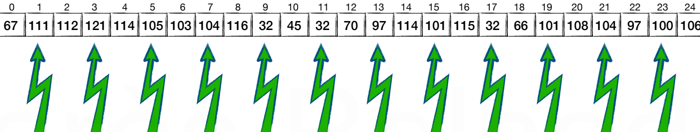
```
dimension = 16777216, resGPU = 4193709.000000, resCPU = 4099894.250000, timeGPU = 21.369000, timeCPU = 51.
dimension = 16777216, resGPU = 4195070.000000, resCPU = 4100987.250000, timeGPU = 20.993000, timeCPU = 50.
dimension = 16777216, resGPU = 4191927.500000, resCPU = 4097727.000000, timeGPU = 36.311000, timeCPU = 50.
dimension = 16777216, resGPU = 4193992.250000, resCPU = 4100070.750000, timeGPU = 20.996000, timeCPU = 50.
dimension = 16777216, resGPU = 4195216.000000, resCPU = 4101382.500000, timeGPU = 20.987000, timeCPU = 50.
```

TP0 : Produit scalaire, V Finale

Reprendre la structure du *sample* `vectorAdd`, avec l'ensemble de ses tests (alloc. mémoire ...), et y implémenter votre version finale du produit scalaire.

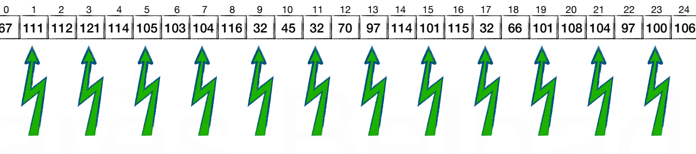
Farès Belhadj

Tris : le tri à bulles (1/6)



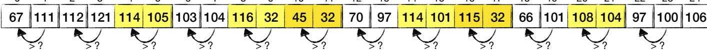
Data de 25 éléments, pour l'instant 1 bloc, 12 threads (int)(25/2) positionnés en idx impair

Tris : le tri à bulles (2/6)

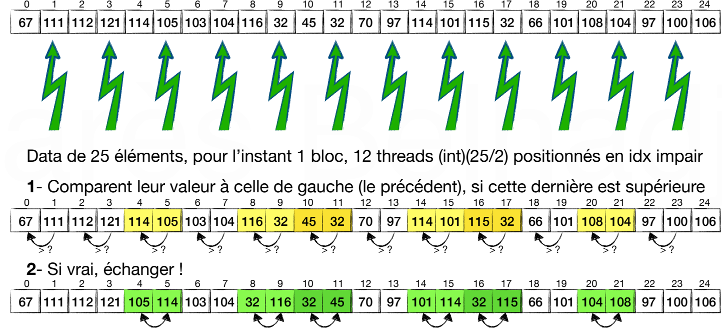


Data de 25 éléments, pour l’instant 1 bloc, 12 threads (int)(25/2) positionnés en idx impair

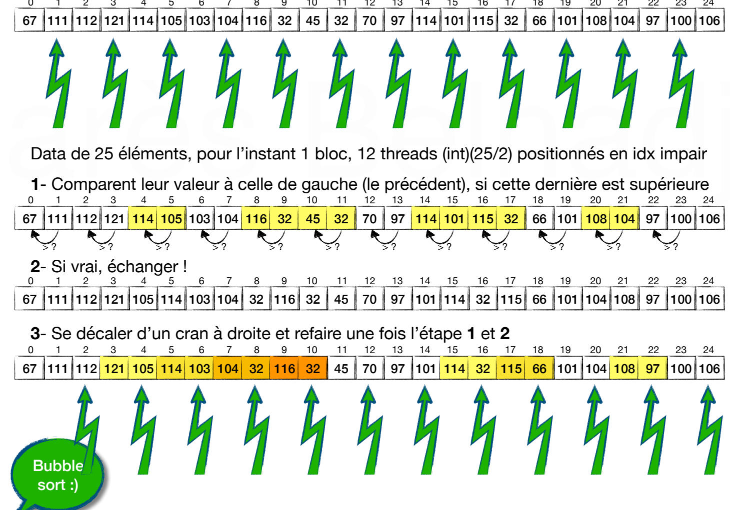
1- Comparent leur valeur à celle de gauche (le précédent), si cette dernière est supérieure



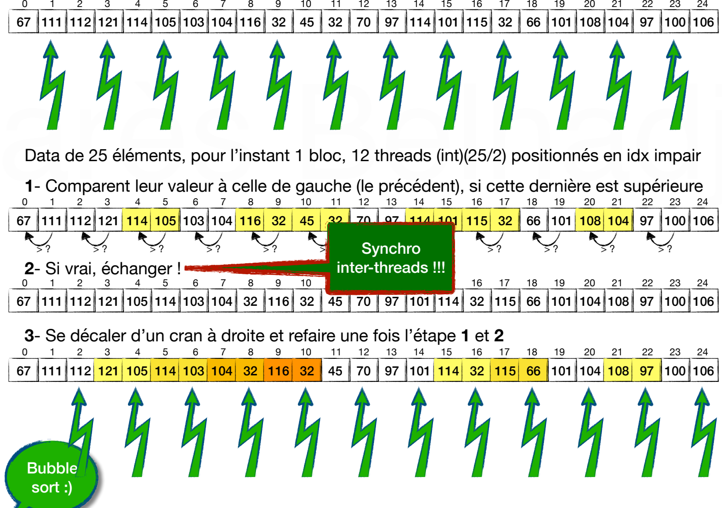
Tris : le tri à bulles (3/6)



Tris : le tri à bulles (4/6)

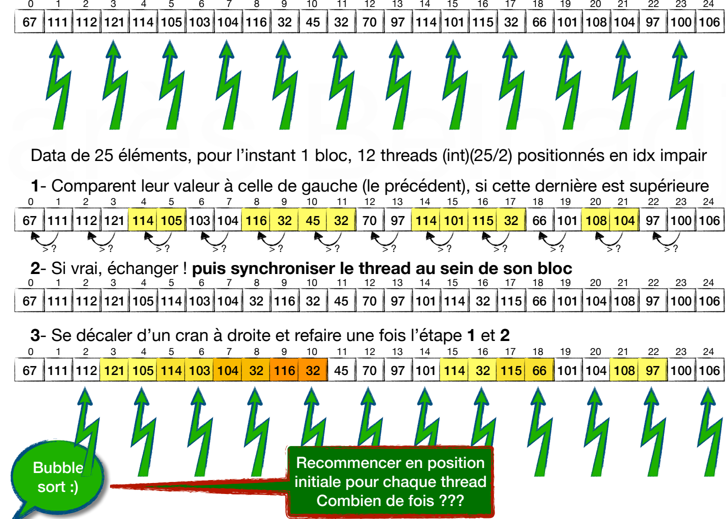


Tris : le tri à bulles (5/6)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
67	111	112	121	114	105	103	104	116	32	45	32	70	97	114	101	115	32	66	101	108	104	97	100	106
67	111	112	121	105	114	103	104	32	116	32	45	70	97	101	114	32	115	66	101	104	108	97	100	106
67	111	112	121	105	114	103	104	32	116	32	45	70	97	101	114	32	115	66	101	104	108	97	100	106

Tris : le tri à bulles (6/6)



TP1 : Implémenter le tri à bulles en GPU (1/3)

Commençons *light* :

- Trions un tableau de `floats` ;
- Écrire une fonction CPU (avec des assertions par exemple) permettant de tester que le tri s'est bien déroulé ;
- Le nombre d'éléments du tableau est inférieur à 1024 pour n'utiliser qu'un seul bloc ;
- Conseil 1 : écrire une fonction de type `__device__`, appelons-la `sub_bsort`, qui prends en paramètre le pointeur vers la donnée, un indice relatif au positionnement du thread et le nombre d'éléments de la donnée, et qui réalise les étapes une et deux des schémas précédents ;
- Conseil 2 (si le premier est suivi) : écrire une fonction de type `__global__`, appelons-la `bsort`, qui prends en paramètre le pointeur vers la donnée et le nombre d'éléments de la donnée, et qui calcule (facile tant qu'un seul bloc) l'indice `i` du thread et appelle **un certain nombre de fois** `sub_bsort`, un coup avec `i`, un coup avec `i + 1` (attention, ça peut déborder ...) ;
- Créer la donnée avec des valeurs aléatoires (allocations, set des valeurs, transferts aller/retour, ...), appeler votre tri GPU en mesurant le temps (incluant les transferts) et tester si la donnée est bien triée ;
- Éventuellement implémenter une version CPU du tri à bulles histoire de comparer les deux.

Farès Belhadj

TP1 : Implémenter le tri à bulles en GPU (2/3)

Passons au multi-blocs :

1. Restons sur un nombre d'éléments sous la forme $N = 2^n$, et inférieur ou égal à 1024 ($n = 10$ au maximum);
2. Un bon nombre de *threads* par bloc est 128;
3. Calculer le bon nombre de blocs et tester;
4. Passons à un N quelconque mais toujours inférieur ou égal à 1024;
5. Revoir si nécessaire le bon nombre de blocs et re-tester;
6. Passons à un N plus grand ... ça ne marche plus?

Farès Belhadj

TP1 : Implémenter le tri à bulles en GPU (3/3)

Version finale :

1. Problème : les blocs ne s'attendent pas entre-eux et la "bulle ne passe pas toujours bien d'un bloc à l'autre" ;
2. Palliatif : faire plus de tours que nécessaire $\rightarrow$ hasardeux ;
3. Il faut forcer les blocs à se synchroniser entre-eux après chaque paire d'échanges. Solution : suite à un appel de kernel les blocs se synchronisent ;
4. Le faire (déplacer la boucle depuis `bsort` vers le CPU) et tester.

Farès Belhadj

CUDA, les textures et les convolutions

Farès Belhadj

Réalisation d'une application filtres-CUDA combinée avec *OpenCV*
Un **sampler de texture** (texture<type, dimensions, cudaReadModeElementType>) :

- peut être 1D, 2D ou 3D ;
- permet d'accéder à une image stockée dans un `cudaArray` à l'aide de coordonnées (espace texture) normalisées ou non ;
- le mode d'accès, en cas de dépassement des bornes de chaque coordonnée, peut être "avec répétition ou enroulé" (`cudaAddressModeWrap`), "coupé" (`cudaAddressModeClamp`), "répété ou enroulé en mirroir" (`cudaAddressModeMirror`) ou enfin "coupé et étalé aux bords" (`cudaAddressModeBorder`) ;
- le filtrage (ou interpolation) peut être linéaire (`cudaFilterModeLinear`) ou "au plus proche" (`cudaFilterModePoint`)
- Exemple :

```
/* Sampler de texture 2D RGBA */
texture<float4, 2, cudaReadModeElementType> tex;
/* paramètres du sampler tex */
tex.addressMode[0] = cudaAddressModeWrap;
tex.addressMode[1] = cudaAddressModeWrap;
tex.filterMode = cudaFilterModeLinear;
tex.normalized = true;
```

CUDA, les textures et les convolutions

Farès Belhadj

La **mémoire texture** est “cachable” (se met en cache), comme la mémoire constante :

- permettant un accès plus rapide à la mémoire (meilleure bande passante) ;
- la proximité spaciale entre les threads (si par exemple dans un block 2D pour une image 2D) permet de mettre en cache une sous-zone de la texture concernant l’ensemble du bloc.

Récupérer l’exemple donné ici.

CUDA, créer une texture

```
/* hData la souce, dData la destination*/
size = Bpp * width * height * sizeof *hData;
checkCudaErrors(cudaMalloc((void **) &dData, size));
/* descripteur d'image en mode RGBA - float */
cudaChannelFormatDesc channelDesc =
  cudaCreateChannelDesc(32, 32, 32, 32, cudaChannelFormatKindFloat);

// Allocate array pour la texture
checkCudaErrors(cudaMallocArray(&cuArray, &channelDesc, width, height));
tex.addressMode[0] = cudaAddressModeWrap;
tex.addressMode[1] = cudaAddressModeWrap;
tex.filterMode = cudaFilterModeLinear;
tex.normalized = true;    // access with normalized texture coordinates
checkCudaErrors(cudaBindTextureToArray(tex, cuArray, channelDesc));
....
// copy image data
checkCudaErrors(cudaMemcpyToArray(cuArray, 0, 0, hData, size, cudaMemcpyHostToDevice));
```

Voir Documentation NVIDIA sur les textures

Et voir le code donné sur le slide précédent.

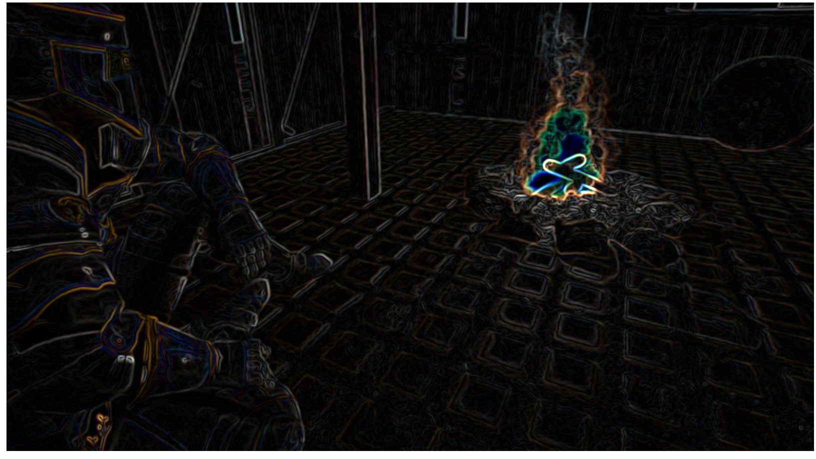
TP2 : les textures et les convolutions (1/8)

Image d'origine

TP2 : les textures et les convolutions (2/8)

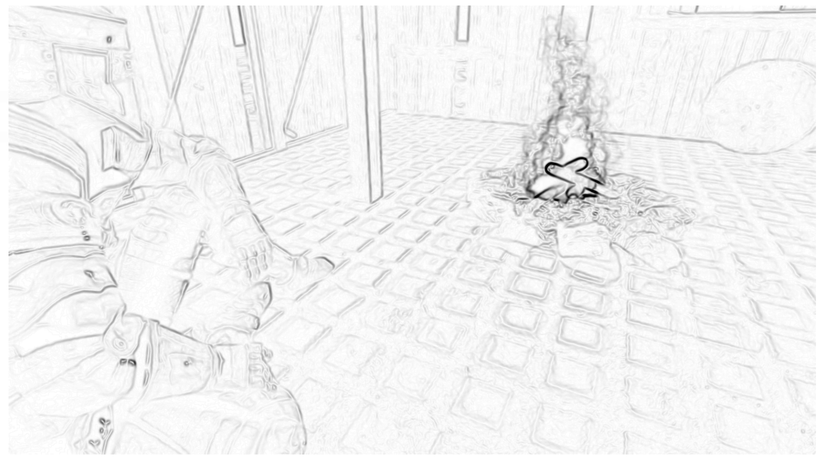
Flou Gaussien

TP2 : les textures et les convolutions (3/8)



Filtre Sobel (consécutif au Gaussien du slide précédent)

TP2 : les textures et les convolutions (4/8)



Luminance du Sobel (du slide précédent)

TP2 : les textures et les convolutions (5/8)

Dispertion ou éparpillement (*scattering*), à partir de l'image d'origine

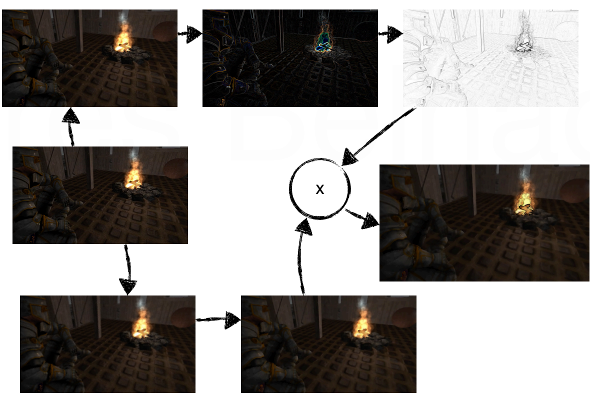
TP2 : les textures et les convolutions (6/8)

Filtre Median de l'image ayant subi un *scattering*

TP2 : les textures et les convolutions (7/8)

Multiplication (Luminance du Sobel) et (Median)

TP2 : les textures et les convolutions (8/8)



L'ensemble du processus

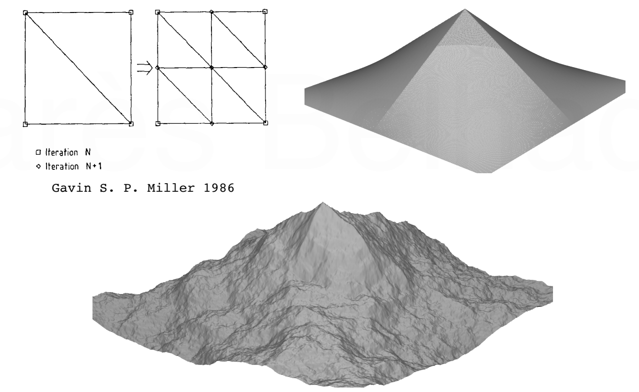
Profondeur ou largeur d'abord en CUDA

Application à la réimplémentation du *Triangle-Edge* ou du *Diamon-Square*

- Algorithmes d'interpolation pour la création de surfaces ;
- Génération de terrains (fractals) en ajoutant du bruit proportionnel à la distance (dans la grille) entre le point interpolé et les extrémités.

Farès Belhadj

Déplacement des milieux en ordre *Triangle-Edge*



En haut à gauche : schéma de subdivision pour l'interpolation. En haut à droite : interpolation sans déplacement des milieux. En bas : interpolation avec déplacement des milieux.

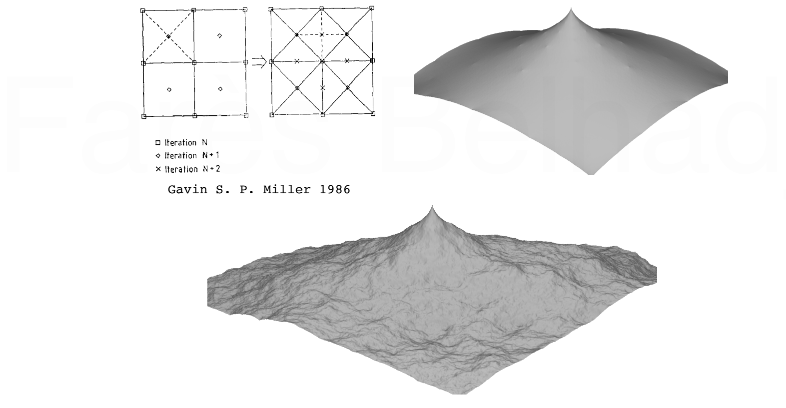
Farès Belhadj

Implémentation séquentielle simplifiée du *Triangle-Edge*

```
void te(unsigned char * data, int x0, int y0, int stride, int sx, int sy, int p) {
  int w_2 = sx >> 1, h_2 = sy >> 1, j;
  int x[9] = { x0, x0 + sx, x0 + sx,      x0, x0 + w_2,  x0 + sx, x0 + w_2,       x0, x0 + w_2 };
  int y[9] = { y0,      y0, y0 + sy, y0 + sy,       y0, y0 + h_2,  y0 + sy, y0 + h_2, y0 + h_2 };
  for(int i = 0, i1 = 1; i < 4; ++i, i1 = (i1 + 1) % 4) {
    if(data[j = y[i + 4] * stride + x[i + 4]]) continue;
    data[j] = ((int)data[y[i] * stride + x[i]] +
               data[y[i1] * stride + x[i1]]) >> 1;
  }
  if(!data[j = y[8] * stride + x[8]]) {
    int v = 0;
    for(int i = 0; i < 4; ++i)
      v += data[y[i] * stride + x[i]];
    data[j] = v >> 2;
  }
  if(w_2 > 1) {
    te(data, x[0], y[0], stride, w_2, h_2, ++p);
    te(data, x[4], y[4], stride, w_2, h_2, p);
    te(data, x[8], y[8], stride, w_2, h_2, p);
    te(data, x[7], y[7], stride, w_2, h_2, p);
  }
}
```

Farès Belhadj

Déplacement des milieux en ordre *Diamond-Square*



En haut à gauche : schéma de subdivision pour l'interpolation. En haut à droite : interpolation sans déplacement des milieux. En bas : interpolation avec déplacement des milieux.

Farès Belhadj

Implémentation CPU simplifiée du *Diamond-Square*

```
#define IN(x, y, w, h) ( (x) >= 0 && (y) >= 0 && (x) < (w) && (y) < (h) )
#define INSERTP(data, x, y, w, h, xpv, ypv, nbp, v) do {              \
    int xp = (xpv), yp = (ypv);                                       \
    if(IN(xp, yp, (w), (h))) { ++(nbp); (v) += (data)[yp * (w) + xp]; } \
  } while(0)

#define DIAMANT do {                                                  \
    int nbp = 0, v = 0;                                               \
    if(data[y * w + x]) continue;                                     \
    INSERTP(data, x, y, w, h, x, y - sh2, nbp, v); /* parent haut   */  \
    INSERTP(data, x, y, w, h, x + sw2, y, nbp, v); /* parent droit  */  \
    INSERTP(data, x, y, w, h, x, y + sh2, nbp, v); /* parent bas    */  \
    INSERTP(data, x, y, w, h, x - sw2, y, nbp, v); /* parent gauche */  \
    data[y * w + x] = v / nbp;                                        \
  } while(0)
void ds(unsigned char * data, int w, int h) {
  int sw1 = w - 1, sw2 = sw1 >> 1;
  int sh1 = h - 1, sh2 = sh1 >> 1;
  while(sw2) {
    for(int y = sh2; y < h; y += sh1) {
      for(int x = sw2; x < w; x += sw1) {
        if(data[y * w + x]) continue;
        data[y * w + x] = ((int)data[(y - sh2) * w + x - sw2] /* parent haut-gauche */ +
                                data[(y - sh2) * w + x + sw2] /* parent haut-droit  */ +
                                data[(y + sh2) * w + x + sw2] /* parent bas-droit   */ +
                                data[(y + sh2) * w + x - sw2] /* parent bas-gauche  */ ) >> 2;
      }
    }
    for(int y0 = 0; y0 < sh1; y0 += sh2)
      for(int y = y0; y < h; y += sh1)
        for(int x = y0 ? 0 : sw2; x < w; x += sw1)
          DIAMANT;
    sw1 = sw2; sw2 >>= 1;
    sh1 = sh2; sh2 >>= 1;
  }
}
```

Farès Belhadj

Aide pour l'insertion d'un déplacement

pour le milieu de manière à controler la rugosité (dimension fractale) du terrain généré

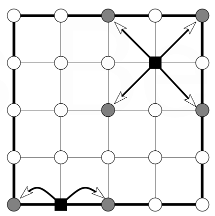
- Ce document permet de comprendre le fonctionnement en une dimension (pour une courbe);
- L'étendre à une surface en considérant que la profondeur est égale au nombre de fois que nous avons coupé le pas en deux, et la distance initiale est égale à la dimension de l'image.
- Faire attention aux distances diagonales.

Farès Belhadj

Profondeur ou largeur d'abord en CUDA

Aborder le problème autrement (voir cet article incluant, en partie (section 2.2), la parallélisation du déplacement des milieux)

Farès Belhadj



■ Exemple de position active à profondeur 2

● Exemple de parents (triangle-edge profondeur 2)

Proposition, faire un pré-calcul en CPU (dépendant de la dimension) permettant de savoir pour chaque position :

1. à quelle profondeur elle doit interpoler sa position (avec garatie que les extrémités (ou "parents") ont déjà été interpolées ;
2. quelles sont les coordonnées des extrémités (ou "parents") qui interviennent dans le calcul d'interpolation.

Projet - pour le 5 décembre

Comparatif CPU/GPU des implémations séquentielles (CPU) et parallèles (CUDA) du déplacement des milieux en ordre *Triangle-Edge* et *Diamond-Square*.

1. Réaliser un programme, qui, en ligne de commande, permet de spécifier <cpu|gpu> <te|ds> <n> <filename.png>. Celui-ci exécute l'implémentation correspondante, génère l'image carrée de dimension $(2^n + 1)^2$, la sauvegarde dans filename.png et produit un print de temps d'exécution de l'algorithme (sans les précalcul s'il y en a, et avec et sans les transferts mémoires de la surface interpolée) ;
2. Produire un graphique comparant les temps de toutes les implémentations de tous les algorithmes pour *n* allant de 3 à 12 ;
3. Préparer une présentation de votre travail (slides en pdf, 10 slides max) ;
4. Envoyer l'ensemble avant le 4 décembre 2019 9h00 ;
5. Venir présenter votre travail le 5 décembre (5 à 6 minutes).

TP3

1. Faire une partie du projet ;
2. Ne concerne que le *Diamond-Square* sans déplacement des milieux et pour une image de 1025×1025 ;
3. Minimum d'accélération attendu : $\times 20$ sur la machine `cuda` ;

Fares Belhadj

Extra : Install Cuda sous Mac OS X High Sierra

Installation CUDA sous macOS 10.13.6, références :

- Guide nvidia, il est préférable de commencer par “3.3 Uninstall” si jamais le Toolkit était déjà installé sur votre système.
- Forum nvidia pour résoudre les problèmes d’installation.

Attention, il y a un problème avec l’installation des drivers MacOS pour nvidia, ils ne permettent pas de faire fonctionner Cuda. La procédure d’install du vrai WebDriver nvidia, décrite sur le forum ci-dessous avait fonctionné l’an dernier (2018) mais n’a pas fonctionné (pour moi) cette année ; j’ai dû avoir recours à un hack décrit sur cette page.

Une fois le WebDriver installé, reboot, m-à-j via “System Preferences” puis “NVIDIA Driver Manager”, reboot ; installer les drivers Cuda, reboot, m-à-j via “System Preferences” puis “CUDA”, reboot ; installer le toolkit CUDA ; TERMINÉ.

M2ISE

Master **I**nformatique des **S**ystèmes **E**mbarqués

Université Paris 8