Comment aborder les GPU?

JM Etancelin, LMAP, UPPA

21 octobre 2022 - Journée MCIA

Les GPU, un sujet d'actualité

- ETSN[1] 2022 : "les GPU, la technologie disruptive du 21ème siècle"
- "OpenCL ?", message du 25/09/2022 sur calcul@listes.math.cnrs.fr
- TOP500:
  - 163 machines GPU
  - 60% puissance de calcul (4 048 908 TFlop / 6 848 441 total)
- Green500 (98.8% des machines ont des GPU)

Dates clés:

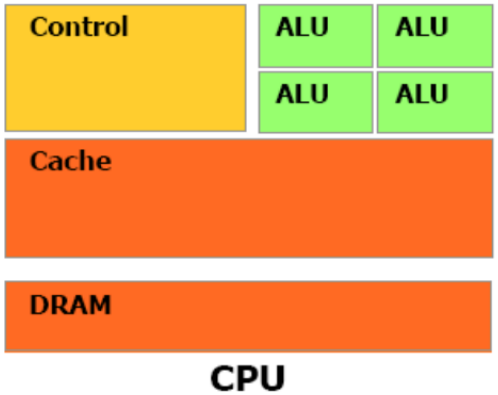
- 1992: OpenGL
- 2007: CUDA (NVIDIA)
- 2008: OpenCL
- 2012: OpenACC
- 2014: SyCL
- 2020: OpenMP-GPUs

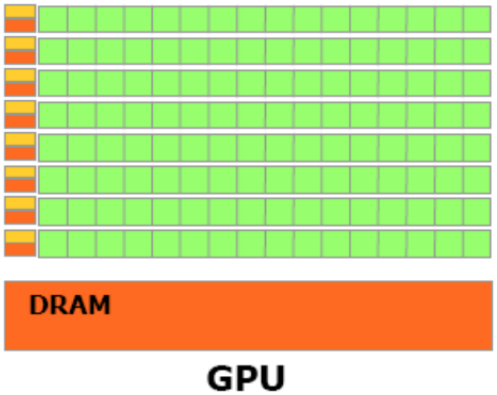
Machines

- JeanZay (idris)
- Adastra (cines)
- curta (MCIA)
- PlaFRIM (inria)
- ...

[1] Ecole Thematique de Simulation Numérique

Qu'est-ce qu'un GPU?



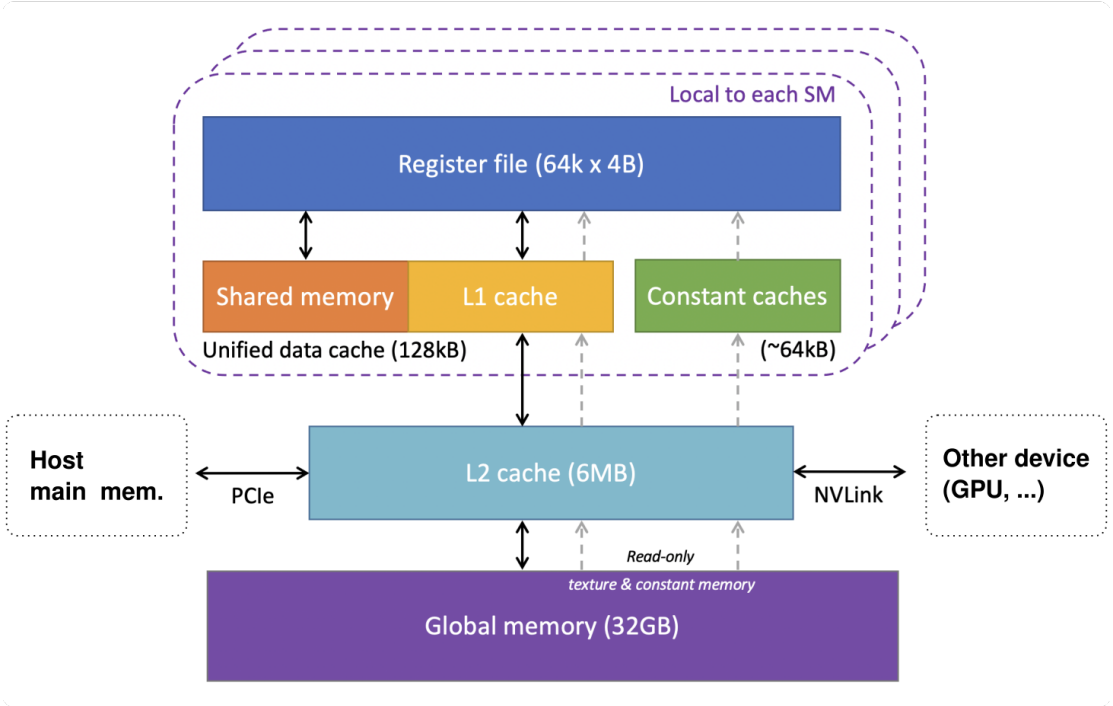


- SIMD : Single Instruction Mutliple Data
- SIMT : Single Instruction Multiple Threads
- Le programme mobilise des threads
- Le matériel gère (ordre, placement, ...) l'exécution les threads

=> Nécessité parallélisme intensif

=> Nécessité de synchronisation

Le GPU dans le système



Données pour le NVIDIA V100

- Différents niveaux de mémoire (physique et logique)
- PCIe bande passante (~30GB/s) comparé à 2TB/s accès mémoire

=> Nécessité de considérer les temps de transferts

Accès aux GPU

2 approches:

- "Intégration":
  - usage de librairies
  - modification légère du code (dépendances, appels de fonction, décorations)
  - gains de performances rapidement accessibles
- "Développement":
  - usage direct du matériel
  - réécriture du code plus ou moins lourde
  - très hautes performances envisageables avec optimisations avancées

Quelle feuille de route vers le GPU?

Quelques constats

- Architectures en évolution plus rapide que les codes
- Variabilité importante d'une génération ou d'un vendeur à l'autre
- Performances dépendantes des caractéristiques du problème

Quelques principes généraux

- le GPU est massivement parallèle
- les performances varient en fonction de la paramétrisation lors déploiement du parallélisme

En pratique ?

Code existant

1. Profilage du code existant
   - Repérage des parties à accélérer
   - Intensité arithmétique / Parallélisation à 1k+ threads / ...
   - Schéma d'utilisation des données
2. Accélération du code
3. Validation (résultats et temps de calculs)
4. Processus itératif : étendre le portage GPU de proche en proche

=> Même si le GPU n'est pas retenu, ces étapes s'appliquent aussi au code CPU.

Nouveau code

1. Identifier précisément les parties parallèles de l'algorithme
2. Déterminer une structure de données adaptée au parallélisme
3. Choisir une approche pour l'implémentation (haut niveau si possible)

=> **Difficulté**: choix pertinent de technologie d'accélération

Exemple: dgemm

$C = \alpha AB + \beta C$, avec A, B et C des matrices rectangulaires et α, β deux réels

- Matériel : machine de TP (uppa)
  - Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz
  - NVIDIA GeForce RTX 3060 (3584 CUDA Cores)
- Logiciels:
  - nvidia sdk 22.5 (cuda 11.7)
  - HIP 4.4
- Contraintes:
  - niveau équivalent de non-optimisation.
  - Utilisation du SDK NVIDIA (cuda, profiler, ...)
  - Focalisation sur l'usage et non les performances obtenues (machine en usage non exclusif, GPU non certifié calcul)
- Calcul simple mais difficile à optimiser

Bibliothèques

- Compilation + édition des liens vers la bibliothèque GPU
  - Adaptation des appels de fonctions (API spécifique)
- Alternative: dans certains cas (nvblas) possibilité de remplacer la librairie lors de l'exécution (`LD_PRELOAD`) sans recompiler.

=> très peu d'efforts à fournir au niveau du code existant

Exemple: Bibliothèque nvBLAS (nvidia)

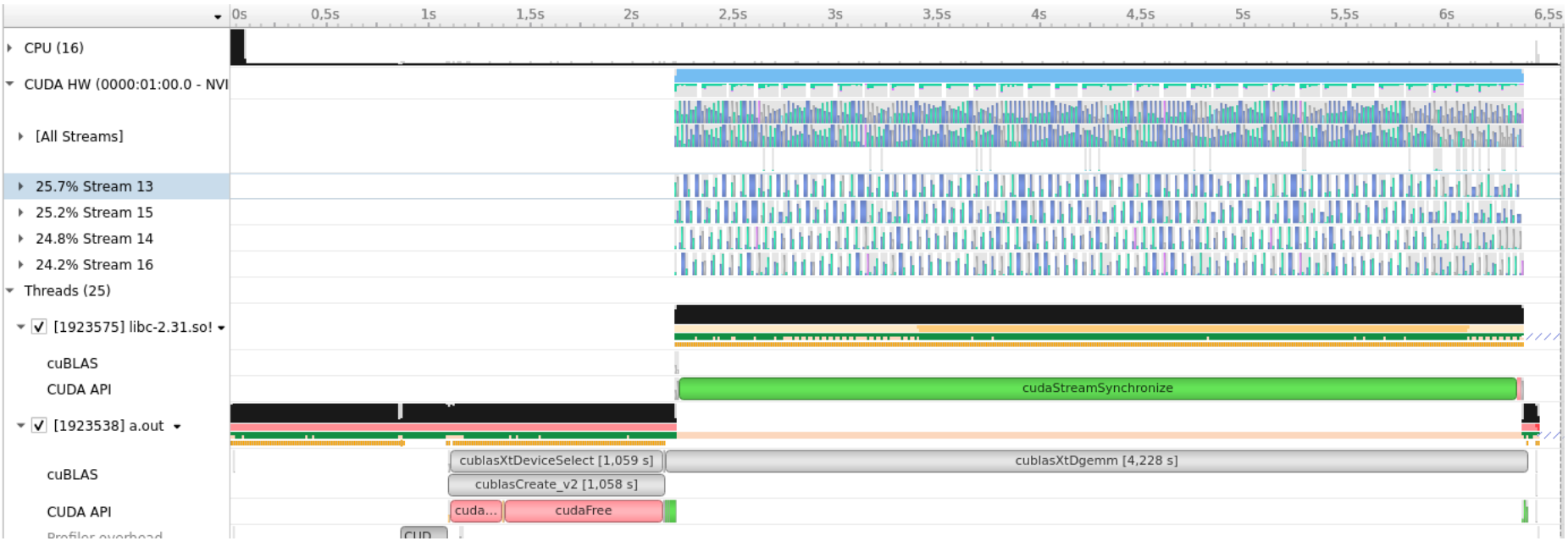
Exemple en langage C:

```
double *A, *B, *C; int m = 8000, k = 6000, n = 8000;
...
A = (double *)malloc( m*k*sizeof( double ));
...
dgemm_(&TRANS, &TRANS, &m, &n, &k,
       &alpha, A, &m, B, &k, &beta, C, &m);
```

```
$> gcc -o c_blas c_blas.c -lblas && ./c_blas
...
Computations completed, in 10.6166 s.

$> LD_PRELOAD=/usr/local/cuda/lib64/libnvblas.so ./c_blas
...
Computations completed, in 5.20307 s.

$> gcc -o nv_blas c_blas.c -L/usr/local/cuda/lib64 -lnvblas && ./nv_blas
...
Computations completed, in 5.14137 s.
```

Exemple:

Profilage:



Pré-chargement de libvnblas.so

Exemple: Octave

```
m = 8000; k = 6000; n = 8000;
A = double( rand(m,k) );
B = double( rand(k,n) );
...
C = A*B;
```

```
$> octave-cli ./mm.m
Elapsed Time = 11.402542
```

```
$> LD_PRELOAD=/usr/local/cuda/lib64/libnvblas.so octave-cli ./mm.m
Elapsed Time =  6.002655
```

Directives de compilation

- Déport du calcul sur gpu pour certaines boucles parallèles
  - À identifier avec un profiler
- Compatibilité avec d'autres directives CPU (OpenMP)
- Conservation d'une unique base de code pour le calcul
  - Ajout éventuel d'informations spécifiques à la gestion de la localité des données. (optimisation)

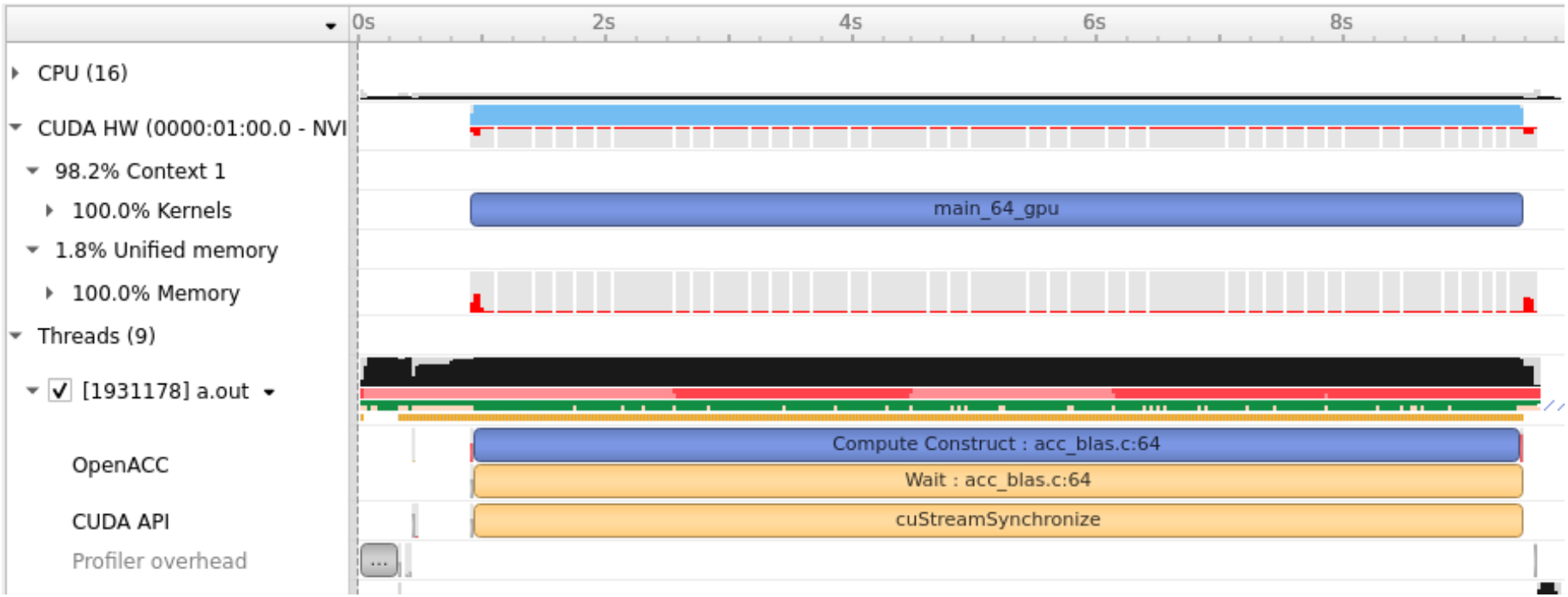
=> Modification "légère" du code existant

OpenACC / OpenMP-GPU

```
#pragma acc data copyin(A[0:m*k],B[0:k*n]) copy(C[0:m*n])
#pragma acc parallel loop tile(32,32)
for(i=0;i<m;i++)
    for(j=0;j<n;j++) {
        C[j+i*n] *= beta;
#pragma acc loop seq
        for(int l=0;l<k;l++)
            C[j+i*n] += alpha*A[l+i*k]*B[j+l*n];
    }
```

```
$> nvc -o acc -acc=gpu -gpu=managed -O3 acc.c && ./acc
...
Computations completed, in 8.57473 s.
```

OpenACC - profile



OpenACC - améliorations

- Pour aller plus loin dans l'exemple:
  - Nécessité de gérer les transferts de données H$\leftrightarrow$D
  - Recouvrement transferts-calculs avec un algorithme par bloc (gros-grain) + asynchronisme
- Pour aller plus loin sur OpenACC:
  - Optimisations de boucles (hiérarchie de threads)
  - Asynchronisme (modifications de l'algorithme)
  - Inter-opérabilité avec CUDA

OpenMP target

- Similaire à OpenACC:

```
#pragma omp target map(to:A[0:m*k],B[0:k*n]) \
                   map(tofrom:C[0:m*n])
#pragma omp teams distribute parallel for
    for(i=0;i<m;i++)
        for(j=0;j<n;j++) {
            C[j+i*n] *= beta;
            for(int l=0;l<k;l++)
                C[j+i*n] += alpha*A[l+i*k]*B[j+l*n];
        }
```

Approche haut niveau

Standard moderne

- C++17, C++20, (C++23, ...)
  - `nvc++ -stdpar=gpu` (nvidia, avec libcu++)
  - GCC avec stdc++ (OpenMP en support)

```
auto r = std::ranges::iota_view(n);
std::transform(std::execution::par, begin(r), end(r),
    [=](auto id){ A[id] = id; });
```

(multi-dimensions `std::ranges::cartesian_product` C++23)

- Fortran 2018, Fortran 202x
  - `nvfortran -stdpar=gpu` (nvidia)
  - `gfortran` (pas encore de support des gpu ?)

```
do concurrent (j=1:n, i=1:m)
  C(i,j) = beta*C(i,j)+alpha*dot_product(A(i,:),B(:,j))
enddo
```

```
Computations completed, in 11.161 s.
```

Approche haut niveau

Précurseurs du standard

SYCL

- À partir du code source SYCL vers ...
  - OpenCL avec `ComputeCpp`
  - OpenCL/CUDA avec Intel-oneAPI
  - OpenMP/ROCm/CUDA avec hipSYCL
  - FPGA avec triSYCL
- Certains supports matériels sont +/- expérimentaux
- Cuda-to-SYCL: existence d'outils de traduction automatique

```
sycl::queue Q{sycl::gpu_selector{}};
int *A = sycl::malloc_shared<int>(N,Q);
Q.parallel_for(N, [=](sycl::item<1> id) { A[id] = id; }).wait();
```

Kokkos

Modèle de programmation C++ visant de manière portable différents types de matériels avec support (CUDA, HIP, SYCL, HPX, OpenMP, ...)

```
Kokkos::parallel_for(N, [=](const int id) { A[id] = id; });
```

Python

- remplacer `numpy` par `cupy` (nvidia)
- remplacer `pandas` par `cudf` (nvidia)

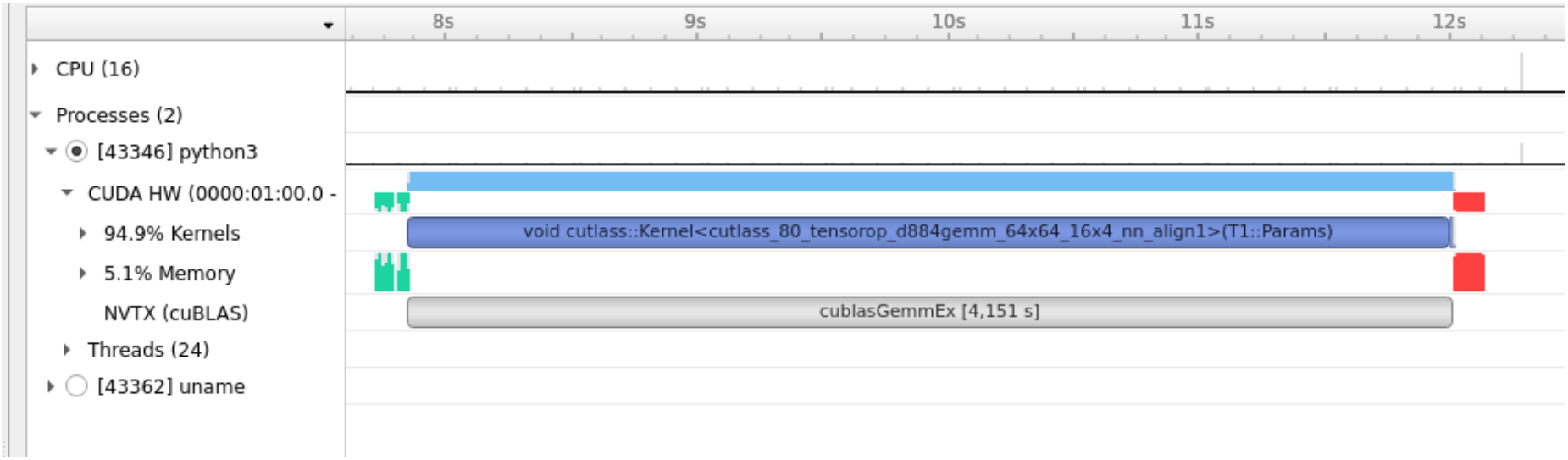
```
import cupy as cp
def dgemm(alpha,A,B,beta, C):
    A_d=cp.asarray(A)
    B_d=cp.asarray(B)
    C_d=cp.asarray(C)
    C[...] = (cp.matmul(A_d,B_d)*alpha + C_d*beta).get()

Computations completed, in 4.456811559037305 s.
```

Python + numba

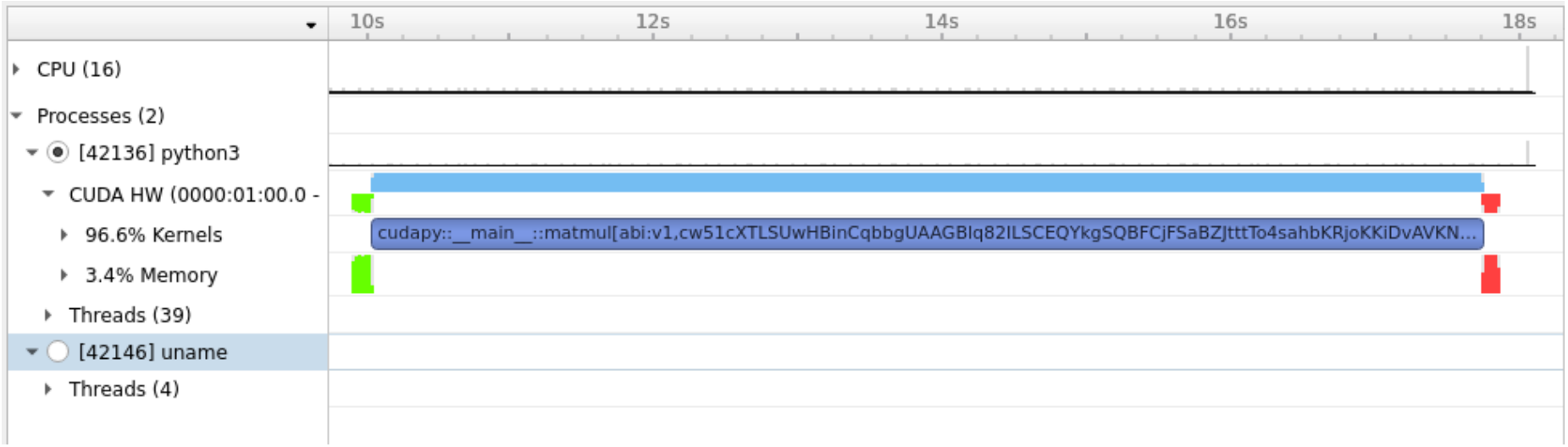
```
from numba import cuda
@cuda.jit
def dgemm(a, A, B, b, C):
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        cij = beta*C[i, j]
        for k in range(A.shape[1]):
            cij += alpha*A[i, k] * B[k, j]
        C[i, j] = cij

Computations completed, in 7.992314503000671 s.
```

Python

cupy



numba



Langage dédié

Interfaces Python

- pyCUDA
- pyOpenCL

=> Nécessité d'écrire un code spécifique GPU + compilation à la volée
Avantage: Code GPU = chaîne de caractères (⇒ Génération de code possible)

Langages compilés

- CUDA
- OpenCL
- HIP

Example: pyCUDA

```
_dgemm = SourceModule("""
  __global__ void dgemm(double *A, double *B, double *C,
                        double alpha, double beta,
                        int m, int k, int n)
  {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < m && col < n) {
      double cij = beta * C[row * n + col] ;
      for (int i = 0; i < k; i++) {
        cij += alpha * A[row * k + i] * B[i * n + col];
      }
      C[row * n + col] = cij;
    }
  }
  """).get_function("dgemm")
...
_dgemm(cuda.In(A), cuda.In(B), cuda.InOut(C),
       np.float64(alpha), np.float64(beta),
       np.int32(m), np.int32(k), np.int32(n),
       block=(32,32,1))
```

Example: pyOpenCL

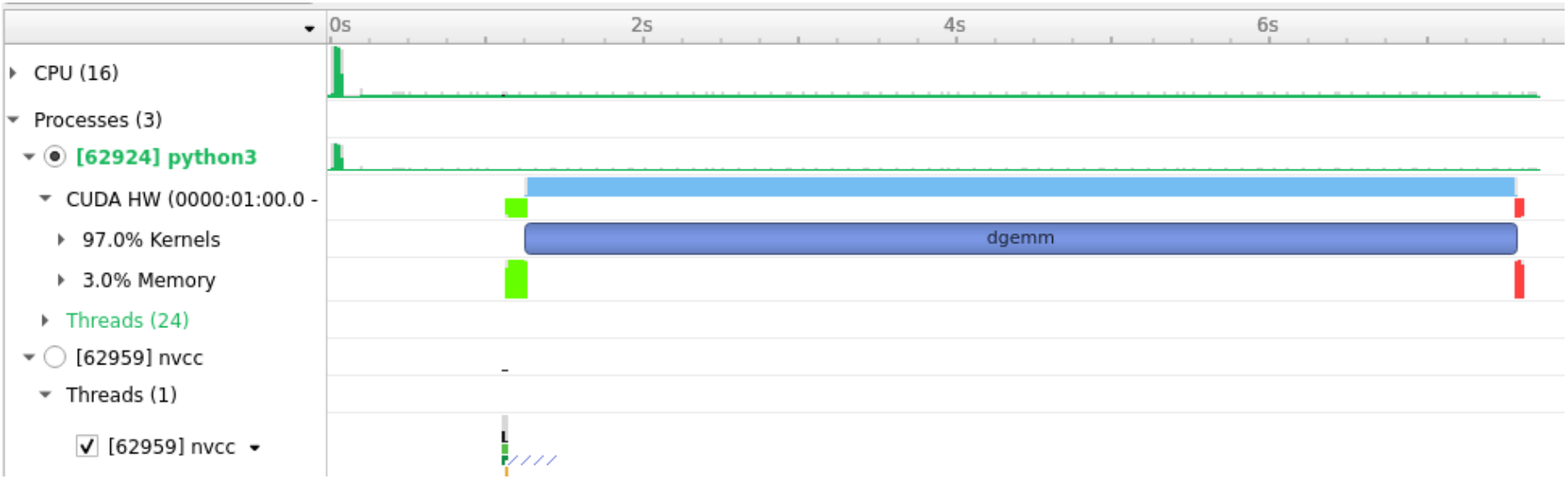
```
prg = cl.Program(ctx, """
  __kernel void dgemm(__global const double *A,
                      __global const double *B,
                      __global double *C,
                        double alpha, double beta,
                        int m, int k, int n)
  {
    int row = get_global_id(1);
    int col = get_global_id(0);
    if (row < m && col < n) {
      double cij = beta * C[row * n + col] ;
      for (int i = 0; i < k; i++) {
        cij += alpha * A[row * k + i] * B[i * n + col];
      }
      C[row * n + col] = cij;
    }
  }
""").build()
```

Example: pyOpenCL (suite)

```
...
    A_d = cl_array.to_device(queue, A)
    B_d = cl_array.to_device(queue, B)
    C_d = cl_array.to_device(queue, C)
    prg.dgemm(queue, C.shape, (32,32),
              A_d.data, B_d.data, C_d.data,
              np.float64(alpha), np.float64(beta),
              np.int32(m), np.int32(k), np.int32(n))
    queue.finish()
    C_d.get(queue, C)

Computations completed, in 8.564110791998246 s.
```

Example: Python

pycuda



pyopencl: profilage non supporté par les outils nvidia (utiliser un outils spécifique TAU, par exemple)

Bilan partiel Python+GPU

cupy / pycuda / pyopencl

- existence d'une interface de tableau (numpy) même pour les données gpu
- opérations éléments-à-éléments (syntaxe ou fonctions des API)
- algorithmes génériques disponibles (réductions, scan, . . .)

cupy

- Code natif python (numpy) inchangé
- Travail en cours de nvidia : cunumeric (version multi-gpu de cupy)

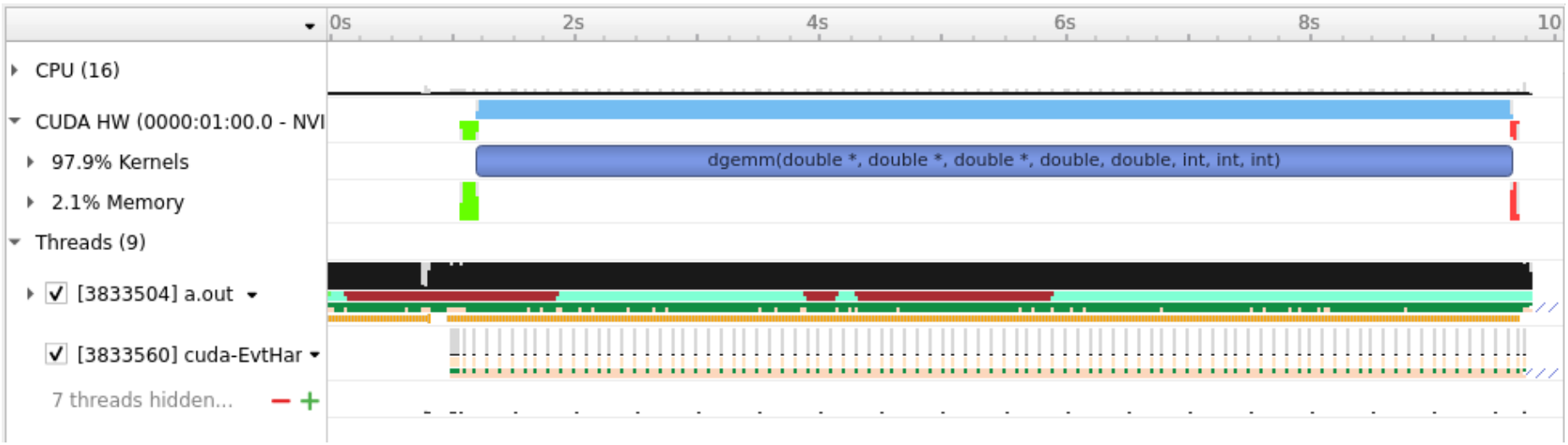
Langage spécifique

CUDA

```
__global__ void dgemm(...) { ... } // identique aux sources pour pycuda
...

    double *A_d, *B_d, *C_d;
    cudaMalloc((void **)&A_d, m*k*sizeof(double));
    cudaMalloc((void **)&B_d, k*n*sizeof(double));
    cudaMalloc((void **)&C_d, m*n*sizeof(double));
    cudaMemcpy(A_d, A, m*k*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, k*n*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(C_d, C, m*n*sizeof(double), cudaMemcpyHostToDevice);
    dim3 block(32,32); dim3 grid(1+(m-1)/block.x,1+(n-1)/block.y);
    dgemm<<<grid,block>>>(A_d, B_d, C_d, alpha, beta, m,k,n);
    cudaMemcpy(C, C_d, m*n*sizeof(double), cudaMemcpyDeviceToHost);

Computations completed, in 8.66002 s.
```

Langage spécifique

CUDA



OpenCL

Similaire à CUDA excepté la gestion explicite de la plateforme, device, file d'exécution.

Langage spécifique

HIP

- C++ API et kernel language pour cibler des GPU nvidia et AMD à partir d'un seul code source.
- Outils permettant de traduire un code CUDA en HIP

Exemple

```
hipify-perl cuda.cu > hip.cpp
hipcc -std=c++11  hip.cpp && ./a.out

...
Computations completed, in 4.50532 s.
```

=> Temps d'exécution identique à la version cuda

Langage spécifique

HIP

- Sur une plateforme nvidia, tout reste en cuda !
- HIP (langage) est très similaire à cuda

```
diff cuda.cu cuda.cu.cpp
0a1
> #include "hip/hip_runtime.h"
8c9
< #include <cuda.h>
---
> #include <hip/hip_runtime.h>
69,74c70,75
<     cudaMalloc((void **)&A_d, m*k*sizeof(double));
<     cudaMalloc((void **)&B_d, k*n*sizeof(double));
<     cudaMalloc((void **)&C_d, m*n*sizeof(double));
<     cudaMemcpy(A_d, A, m*k*sizeof(double), cudaMemcpyDefault);
<     cudaMemcpy(B_d, B, k*n*sizeof(double), cudaMemcpyDefault);
<     cudaMemcpy(C_d, C, m*n*sizeof(double), cudaMemcpyDefault);
---
>     hipMalloc((void **)&A_d, m*k*sizeof(double));
>     hipMalloc((void **)&B_d, k*n*sizeof(double));
>     hipMalloc((void **)&C_d, m*n*sizeof(double));
>     hipMemcpy(A_d, A, m*k*sizeof(double), hipMemcpyDefault);
>     hipMemcpy(B_d, B, k*n*sizeof(double), hipMemcpyDefault);
>     hipMemcpy(C_d, C, m*n*sizeof(double), hipMemcpyDefault);
76,77c77,78
<     dgemm<<<grid,block>>>(A_d, B_d, C_d, alpha, beta, m,k,n);
<     cudaMemcpy(C, C_d, m*n*sizeof(double), cudaMemcpyDefault);
---
>     hipLaunchKernelGGL(dgemm, grid, block, 0, 0, A_d, B_d, C_d, alpha, beta, m,k,n);
>     hipMemcpy(C, C_d, m*n*sizeof(double), hipMemcpyDefault);
```

Bilan de l'exemple

Langage/Bibliothèque	#Lignes de code	Temps calcul[2]
C + OpenBLAS (cpu)	64	10.9869
C + nvBLAS	0 (64 code CPU)	5.01542
Octave (cpu)	11	11.197891
Octave + nvblas	0 (11 code CPU)	5.645218
C + OpenACC	3 (64 code CPU)	8.54927
C + OpenMP	3 (64 code CPU)	-
Fortran 2018	1 (35 code CPU)	10.907
cupy	1 (27 code CPU)	4.50632
numba	44	8.08281
pyCUDA	47	6.56785
pyOpenCL	56	6.64471
C + CUDA	87	8.80147
C + OpenCL	186	8.58346
HIP	0 (87 C+CUDA)	-

[2] À titre indicatif (non optimisé, non moyenné)

Conclusions

Environnements de programmation:

- NVIDIA HPC SDK (cuda + profiler + dégogeurs + libraries + openacc + opencl + ...)
- AMD ROCm (hip + profiler + librairies + hipify + opencl + ...)

Trajectoire d'approche des GPU:

=> privilégier une approche haut niveau pour le prototypage et le portage

- Code existant:
  - python: assez direct avec cupy
  - C/C++/Fortran: openmp target (portabilité)
- Nouveau code:
  - python + numba/cupy
  - C++/Fortran: standards modernes

Merci pour votre attention

Sources des exemples utilisés ici

- git.univ-pau.fr:jmetancelin/gpu-tech-review.git

Références (liens)

- Developing HPC Applications with Standard C++, Fortran, and Python
- A Deep Dive into the Latest HPC Software
- SYCL introduction
- Pourquoi kokkos?
- HIP porting guide
- Tutoriels pyCUDA et pyOpenCL