Calcul Québec

Exercices: https://github.com/calculquebec/cq-formation-openacc
Tutoriel écrit: https://docs.alliancecan.ca/wiki/OpenACC_Tutorial/fr

Programmation GPU facile avec OpenACC

Par : Maxime Boissonneault

Nos membres et partenaires

Partenaire de l'

Alliance de recherche numérique du Canada

Partenaires connectivité

Partenaires financiers

Qui suis-je

Maxime Boissonneault

- Physique (Ph. D.)
- Mesure de qubits supraconducteurs avec Alexandre Blais (UdeS)
- Utilisateur des premières grappes de calcul à l'UdeS
- À Calcul Québec à ULaval depuis 2012
- Team lead, Équipe nationale de soutien à la recherche, Alliance de recherche numérique du Canada et Calcul Québec

Rappel des prérequis

Être capable de vous connecter à une grappe de calcul Linux et d'interagir avec celle-ci (éditer des fichiers, naviguer dans les répertoires) en ligne de commande et avoir une connaissance de base du langage C.

Plan de cours

- Introduction aux architectures d’accélérateurs;
- Profiler le code existant et extraire des informations du compilateur;
- Exprimer le parallélisme du code avec des directives OpenACC;
- Exprimer les transferts de données;
- Optimiser les boucles.

Préparation

Connexion à la plateforme

URL	https://p2-ecole.calculquebec.cloud/ https://p-ecole.calculquebec.cloud/

Server Options

Grappe p2

Reservation

None

Partition

nodegpu

Account

def-sponsor00

Time (hours)

4,0

Number of cores

8

Memory (MB)

14000

Enable core oversubscription? Recommended for interactive usage

GPU configuration

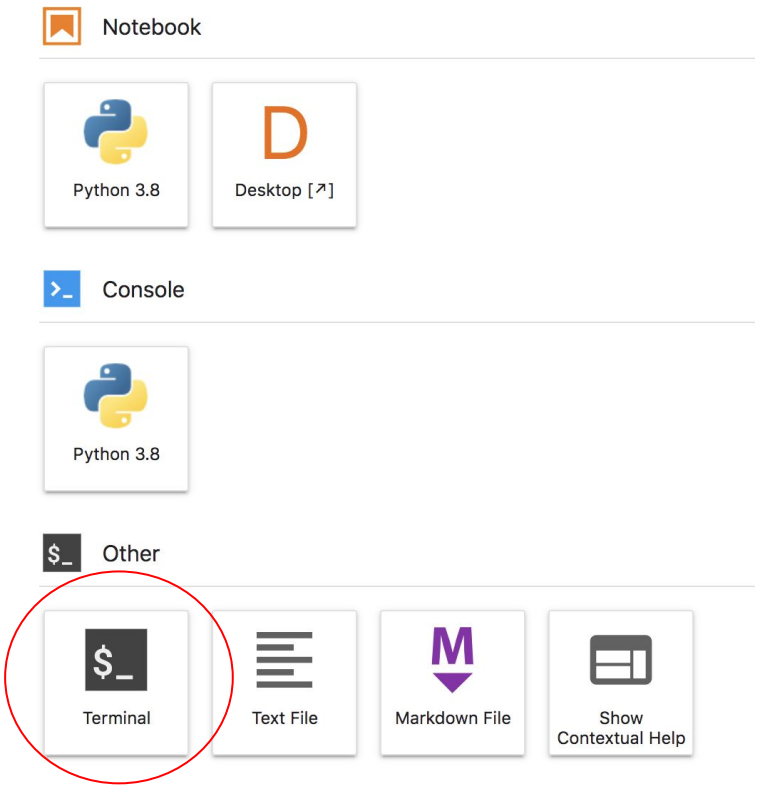
1 x 2G.10GB

User interface

JupyterLab

Ouvrir un terminal et un Desktop

Grappe p2



Grappe p

Server Options

Reservation

None

Partition

gpu-node

Account

def-sponsor00

Time (hours)

4,0

Number of cores

4

Memory (MB)

4096

☐ **Enable core oversubscription?** Recommended for interactive usage

GPU configuration

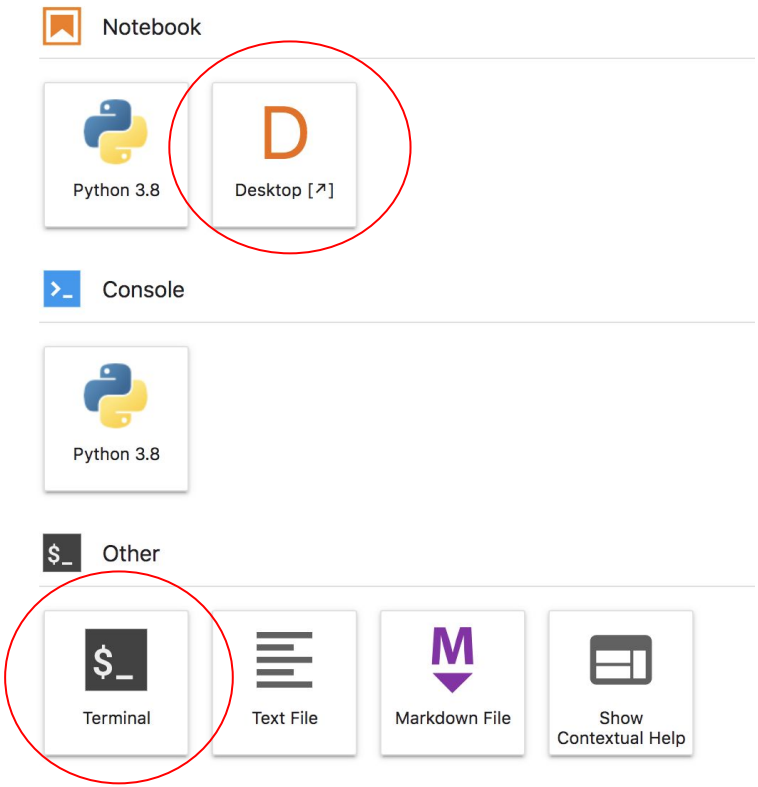
1 x GPU

User interface

JupyterLab

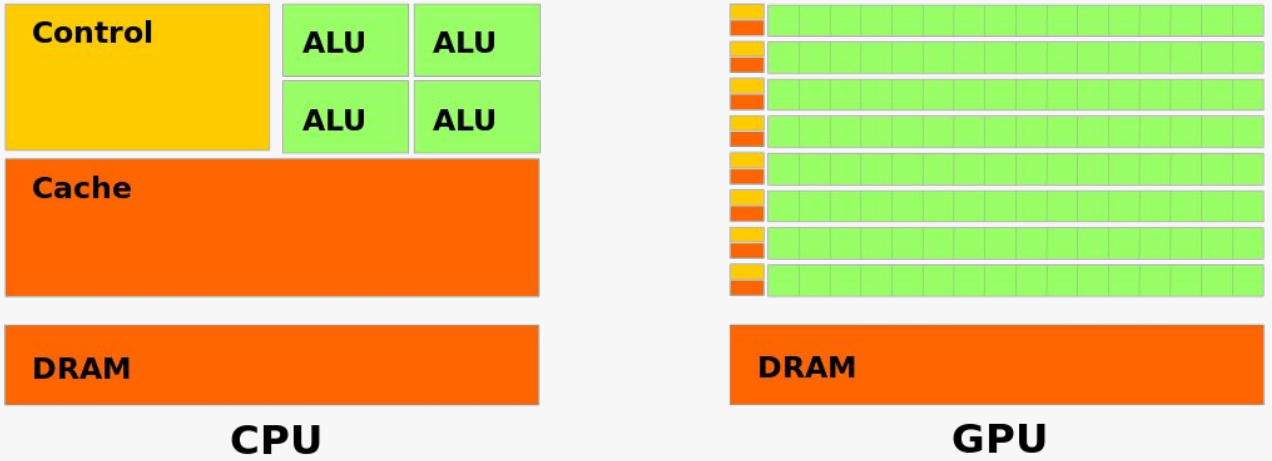
Ouvrir un terminal et un Desktop

Grappe p



Introduction

CPU vs GPU



ALU = Arithmetic and Logic Unit - The workhorse

Vitesse vs rendement

Vitesse

Rendement

Quand porter un code sur GPU ?

Quand porter sur/utiliser un GPU ?

- Code A:
  - 1h: charge des données
  - 6h: calcul parallélisé sur 10 coeurs
  - 1h: écrit les résultats
- Code B:
  - 10min: initialisation + écriture de résultats
  - 5min: calcul séquentiel non parallélisable
  - 4h: calcul séquentiel parallélisable
- Code C:
  - 5min: initialisation + écriture de résultats
  - 10h: calcul séquentiel parallélisable

Comment déterminer si ça vaut la peine ?

Comparaison de prix

- Noeud CPU de Béluga: ~10k$
  - 40 coeurs de calcul
- Noeud GPU de Béluga: ~50k$
  - 40 coeurs de calcul + 4 GPUs V100
- Pour que l'utilisation de GPU soit pertinente, il faut
  - 1 V100 + 10 CPU > **1.25x** la performance de 40 CPU
  - 1 V100 + 10 CPU > **5x** la performance de 10 CPU
  - 1 V100 + 10 CPU > **50x** la performance de 1 CPU

Quand porter sur/utiliser un GPU ?

- Code A:
  - 1h: charge des données
  - 6h: calcul parallélisé sur 10 coeurs
  - 1h: écrit les résultats
  - Total (sur 10 coeurs): 8 heures

Quand porter sur/utiliser un GPU ?

- Code A:
  - 1h: charge des données
  - 6h: calcul parallélisé sur 10 coeurs
  - 1h: écrit les résultats
  - Total (sur 10 coeurs): 8 heures
- Gain maximum: 4x (8/2)
  - => Non pertinent

Quand porter sur/utiliser un GPU ?

- Code B:
  - 10min: initialisation + écriture de résultats
  - 5min: calcul séquentiel non parallélisable
  - 4h: calcul séquentiel parallélisable
  - Total (sur 1 coeur): 4h15
  - Temps minimum: 15min

Quand porter sur/utiliser un GPU ?

- Code B:
  - 10min: initialisation + écriture de résultats
  - 5min: calcul séquentiel non parallélisable
  - 4h: calcul séquentiel parallélisable
  - Total (sur 1 coeur): 4h15
  - Temps minimum: 15min
- Gain maximum: 17x
  - => Non pertinent

Quand porter sur/utiliser un GPU ?

- Code C:
  - 5min: initialisation + écriture de résultats
  - 10h: calcul séquentiel parallélisable
  - Total (sur 1 coeur): 10h5min
  - Temps minimum: 5min

Quand porter sur/utiliser un GPU ?

- Code C:
  - 5min: initialisation + écriture de résultats
  - 10h: calcul séquentiel parallélisable
  - Total (sur 1 coeur): 10h5min
  - Temps minimum: 5min
- Gain maximum: 121x
  - => Pertinent

Quand porter sur/utiliser un GPU ?

Inversons la question:

- Si j'ai 10 minutes de temps non parallélisable, combien de temps une tâche doit durer pour que ce soit pertinent de la porter sur 1 GPU + 10 CPUs ?
  - Si elle est présentement séquentielle ?
  - Si elle est présentement parallélisée sur 10 coeurs ?
  - Si elle est présentement parallélisée sur 40 coeurs ?

Quand porter sur/utiliser un GPU ?

Inversons la question:

- Si j'ai 10 minutes de temps non parallélisable, combien de temps une tâche doit durer pour que ce soit pertinent de la porter sur 1 GPU + 10 CPUs ?
  - Si elle est présentement séquentielle ?
    - 50 x 10 minutes = 8.3 heures
  - Si elle est présentement parallélisée sur 10 coeurs ?
    - 5 x 10 minutes = 50 minutes
  - Si elle est présentement parallélisée sur 40 coeurs ?
    - 1.25 x 10 minutes = 12.5 minutes

Quand porter sur/utiliser un GPU ?

- **Attention:**
- Les calculs précédents supposent que le portage sur GPU permet de **complètement éliminer** ce temps de calcul. En pratique, la portion du code qui est portée sur GPU prend un temps fini. Ce sont des **minimums absolus**.
-
- Ces calculs vont aussi dépendre des générations de matériel.

Quand porter sur/utiliser un GPU ?

- **<u>À retenir</u>**: Les GPUs peuvent être très rapides, mais ils sont aussi très dispendieux. Il est nécessaire d'avoir une très grande accélération du programme **complet** pour qu'il soit pertinent d'utiliser des GPUs.

Optimiser un code

1. Profiler
2. Identifier les goulots
3. Optimiser les goulots
4. Valider les résultats
5. Recommencer

Porter un code vers GPU

1. Profiler
2. Identifier les goulots parallélisables
3. Porter les goulots vers GPU
    a. Exprimer le parallélisme au compilateur
    b. Optimiser les transferts de données
    c. Optimiser les boucles
4. Valider les résultats
5. Recommencer

OpenMP vs OpenACC

OpenMP

- Supporte les accélérateurs depuis la version 4.0
- Supporté par
  - Intel
  - GCC
  - CLang
  - NVHPC (précédemment Portland Group)
- Langage prescriptif

OpenACC

- Créé pour les accélérateurs
- Supporté par
  - NVHPC (précédemment Portland Group)
  - GCC (versions 5+, à divers degrés)
  - LLVM/CLang
- Intel n’a pas l’intention de supporter le standard
- Langage descriptif

OpenMP - Langage prescriptif

1. Compilateur fait ce qui est demandé, peu importe si c’est optimal ou non
2. Programmeur responsable de la validité du code
3. Programmeur responsable d’optimiser pour chaque unité de calcul

OpenACC - Langage descriptif

1.

1. Indications au compilateur, qui optimise par lui-même
2. Compilateur responsable de la validité du code
3. Compilateur responsable d'optimiser pour chaque unité de calcul

4. OpenACC peut aussi être prescriptif si nécessaire

Profiler un code

nvprof, nvvp

1. nvprof
   a. Profileur de NVidia. Spécialisé GPU. Ligne de commande
2. nvvp
   a. Profileur graphique de NVidia. Analyse poussée du comportement GPU

Instructions de compilateur

- -Minfo => Génère des informations
  - all => toutes les infos de base
  - intensity => intensité de calcul
  - ccff => génère des fichiers d'information
  - accel => infos d'accélérateurs
- -acc=gpu => Spécifie un target
  - -acc -gpu=managed => génère du code pour un GPU Tesla avec mémoire gérée automatiquement

Exemple #1 - Obtenir des infos

Ajout de directives

Directives OpenACC

```
1 #pragma acc kernels
 2 {
 3  for (int i=0; i<N; i++)
 4  {
 5    x[i] = 1.0;
 6    y[i] = 2.0;
 7  }
 8
 9  for (int i=0; i<N; i++)
10  {
11    y[i] = a * x[i] + y[i];
12  }
13 }
```

Boucles vs kernels

Loop	Kernels
1 **for** (int i=0; i<N; i++) 2 { 3 C[i] = A[i] + B[i]; 4 }	1 void loopBody(A,B,C,i) 2 { 3 C[i] = A[i] + B[i]; 4 }
Calculate 0 - N in order	Each compute core calculates one value of i.

Exemple #2 - directive *kernels*

Fausses dépendances

- *Pointer aliasing (problème non-existant en Fortran)*
  - Deux pointeurs peuvent pointer vers la même mémoire
  - Empêche certaines optimisations

__restrict

- Promesse au compilateur que les pointeurs ne pointent pas vers la même mémoire
  - Comportement indéfini si la promesse est brisée
- Bonne pratique en général, même sans parallélisation

```
1 double *__restrict Acoefs=A.coefs;
2 double *__restrict xcoefs=x.coefs;
3 double *__restrict ycoefs=y.coefs;
```

Directive loop independent

```
1 #pragma acc kernels
2 {
3 #pragma acc loop independent
4 for (int i=0; i<N; i++)
5 {
6  C[i] = A[i] + B[i];
7 }
8 }
```

Retour à l’exemple

Directive parallel loop

```
1 #pragma acc parallel loop
2 for (int i=0; i<N; i++)
3 {
4   C[i] = A[i] + B[i];
5 }
```

- Prescriptive
  - Compilateur ne fait *que* ce qui est demandé
  - Indépendance des itérations implicite

Directive parallel loop

```
1 #pragma acc parallel loop
 2  for(int i=0;i<num_rows;i++) {
 3    double sum=0;
 4    int row_start=row_offsets[i];
 5    int row_end=row_offsets[i+1];
 6 #pragma acc loop reduction(+:sum)
 7    for(int j=row_start;j<row_end;j++) {
 8      unsigned int Acol=cols[j];
 9      double Acoef=Acoefs[j];
10      double xcoef=xcoefs[Acol];
11      sum+=Acoef*xcoef;
12    }
13    ycoefs[i]=sum;
14  }
```

Exercice (15 minutes)

1. Modifiez les fonctions matvec, waxpby, et dot pour utiliser OpenACC. Vous pouvez utiliser soit la directive *kernels* ou la directive *parallel loop*. Les répertoires *step1.** contiennent la solution.

2. Modifiez le Makefile pour ajouter *-acc -gpu=managed* et *-Minfo=accel* à vos options de compilateur.

Exprimer les transferts de données

Directive data

```
1 #pragma acc data
2 {
3 #pragma acc parallel loop ...
4 #pragma acc parallel loop
5 ...
6 }
```

- Défini une zone dans laquelle les données restent sur le GPU, partagé par tous les kernels

Clauses

Copie:

- copyin(list)
- copyout(list)
- copy(list)

Création/suppression:

- create(list)
- delete(list)

Ne rien faire

- **present(list)**
  - **Toujours préférer *present* lorsque possible**

Transfert de tableaux

```
#pragma acc data copyin(a[0:nelem])
copyout(b[s/4:3*s/4])
```

Directives enter data/exit data

Utiles en C++ pour les constructeurs et destructeurs dans les classes, ou lorsque l'allocation et la désallocation sont séparées

```
1 class Matrix { Matrix(int n) {
2         len = n;
3         v = new double[len];
4         #pragma acc enter data create(v[0:len])
5 }
6 ~Matrix() {
7         #pragma acc exit data delete(v)
8 };
```

Retour à l’exemple

Directive update

```
1 void initialize_vector(vector &v,double val) {
2   for(int i=0;i<v.n;i++)
3       v.coefs[i]=val;   // '''Updating the vector on the
CPU '''
4   #pragma acc update device(v.coefs[:v.n])    //
'''Updating the vector on the GPU'''
6 }
```

Optimiser les boucles

Niveaux de parallélisme OpenACC vs CUDA

- OpenACC vector => CUDA threads
  - Instruction simple sur plusieurs données (SIMD). Effectuées même si les données n'existent pas.
- OpenACC worker => CUDA warps
  - Chaque *worker* exécute un vecteur
- OpenACC gang => CUDA thread blocks
  - Plusieurs *worker*s.
  - Partage de ressources.
  - Plusieurs *gang*s sont complètement indépendantes

Correspondance approximative.

Clauses de la directive *loop*

- gang
- worker
- vector
- seq => séquentielle (pour débogage)

Plusieurs clauses applicables à la même boucle, mais doivent être en ordre (gang, worker, vector)

Clauses de la directive *parallel loop*

- num_gangs
- num_workers
- vector_length

Clauses de la directive *loop*

- device_type
  - Spécifie le type d'accélérateur pour lequel la clause s'applique.

```
device_type(nvidia) vector_length(256) \
device_type(radeon) vector_length(512) \
vector_length(64)
```

Taille de parallélisme

Chaque niveau a une taille. Exemple:

```
worker(32) vector(32)
```

Créera 32 workers qui calculeront des vecteurs de taille 32.

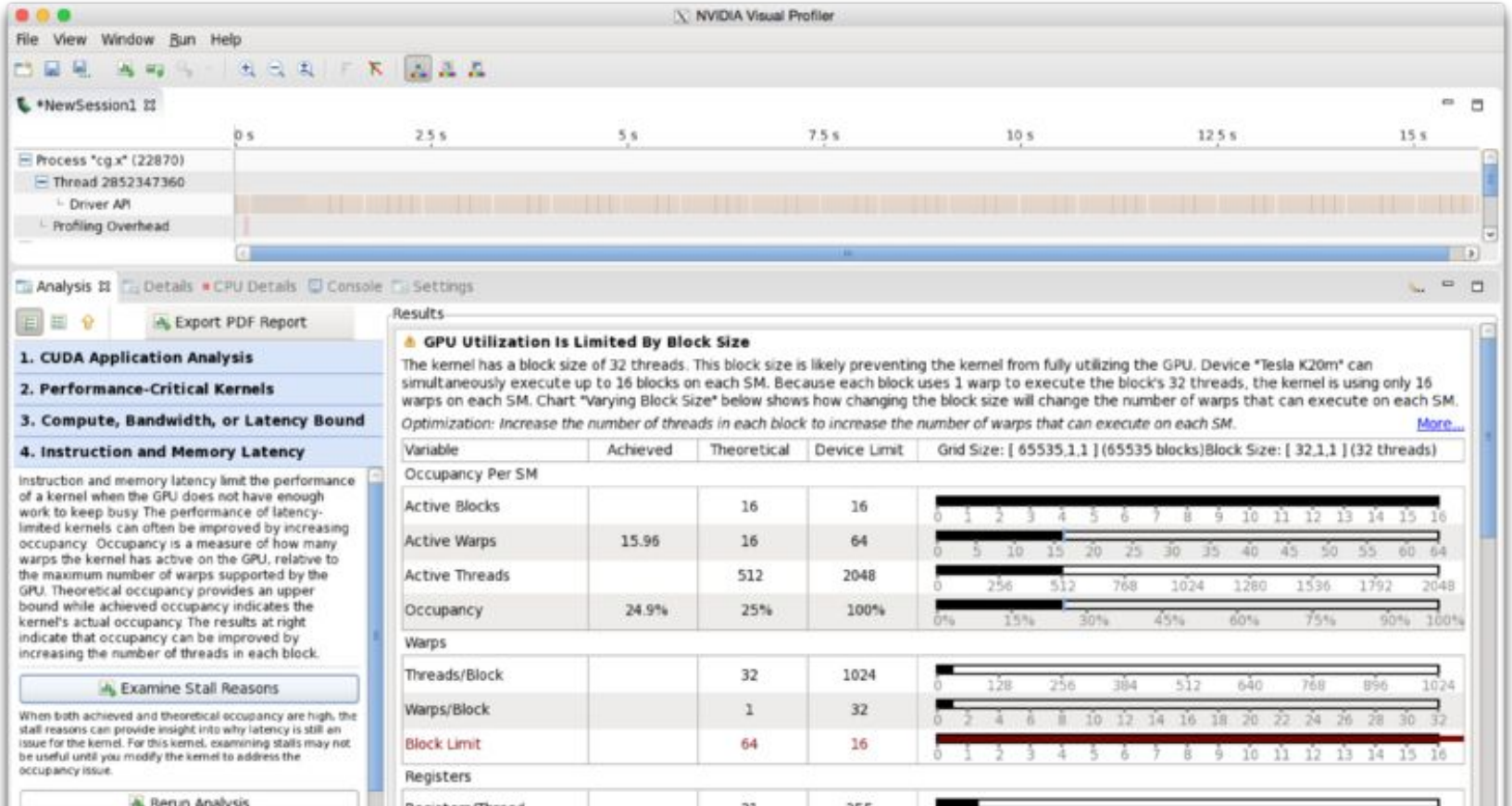
Limitations de taille pour NVidia

- vector => multiple de 32, max 1024
- gang size = worker size x vector size
  - gang size <= 1024

Retour à l’exemple

Analyse guidée avec *nvvp*

Analyse guidée avec nvvp



Autres clauses d’optimisation

- collapse(N)
  - Fusionne les N prochaines boucles imbriquées
- tile(N,[M,...])
  - Réparti les itérations d’une boucle en “tuiles”