Outils pour l'analyse du parallélisme et des noyaux de calcul

EI-SE5 – Calcul haute performance

Adrien Cassagne

Le 11 octobre 2022

Table des matières

Objectifs de ce cours

- Revue des principaux modèles de parallélisme
  - *Single Instruction Multiple Data*
  - *Multi-threading*
  - *Multiple Program Multiple Data*
  - *Single Instruction Multiple Threads*
- Analyse des codes parallèles
  - Temps de restitution
  - Accélération ou *speedup*
  - Loi d'Amdahl
  - Efficacité
  - Passage à l'échelle ou *scalability*
- Analyse de la performance d'un noyau de calcul
  - Opérations par seconde
  - Performance crête
  - Intensité arithmétique et opérationnelle
  - Le modèle *roofline*

Table des matières

2 Modèles d'exécution parallèle

Constat

Nous avons vu dans les cours précédents que toutes les architectures modernes exploitent le parallélisme.

- Impossible de continuer d'augmenter les fréquences (consomme trop d'énergie, inefficace → bulles dans le pipeline)
- Solution : augmenter le nombre de cœurs à fréquence constante
- Le parallélisme est partout : supercalculateurs, ordinateurs, smartphones, montres connectées, consoles de jeu, télévisions, ...
- **Peut-on encore programmer sans tenir compte du parallélisme ?**

Modèle d'exécution Single Instruction Multiple Data

- Vient directement de la taxonomie de Flynn (1966)
- Une même instruction est appliquée à plusieurs données
- Modèle transposable à un cœur CPU et parfois à un cœur GPU
- Se programme généralement en assembleur
  - Ou plutôt avec des fonctions intrinsèques pour ne pas avoir à gérer l'allocation des registres
- Quelques jeux d'instructions SIMD : SSE, AVX, NEON, SVE

Modèle d'exécution multi-threads

- Plusieurs fils d'exécution au sein d'un même processus (IBM OS/360, 1967)
- **La mémoire est partagée** entre les différents fils d'exécution
- Modèle utilisé pour programmer plusieurs cœurs CPU d'un même ordinateur
- Se programme généralement avec des bibliothèques
- Parfois directement avec des langages dédiés (ex. Cilk)
- Quelques bibliothèques *multi-threads* : threads POSIX, OpenMP, Threading Building Blocks (TBB)

Modèle d'exécution Multiple Program Multiple Data

- Plusieurs exécutables exécutés sur des données différentes
- Ou plutôt, plusieurs instances du même programme lancées en parallèle
- **La mémoire est distribuée** entre les différents processus
- Modèle utilisé pour programmer plusieurs cœurs CPU d'un même ordinateur ou pour communiquer entre différents nœuds d'un supercalculateur
  - **Modèle de prédilection dans le calcul haute performance**
- Se programme généralement avec des bibliothèques
- Un standard bien établi : *Message Passing Interface* (MPI, 1991)
- Peut être vu comme une surcouche aux *sockets* (réseau)
- Beaucoup d'implémentations : OpenMPI, MPICH, Intel MPI, IBM MPI, HP MPI, ...

Modèle d'exécution Single Instruction Multiple Threads

- Un mix entre le modèle SIMD et le modèle *multi-threads* (Nvidia, 2007)
- En SIMT, il y a des *Work-items* qui font tous la même opération (comme en SIMD)
- Plusieurs *Work-items* sont rassemblés dans des *Wavefronts* qui correspondent plus à des threads dans le modèle *multi-threads*
- **Modèle utilisé pour programmer les GPU** et parfois les CPU/FPGA
- Se programme généralement avec des langages dédiés (CUDA, OpenCL, Metal, ...)

Modèles d'exécution parallèles : récapitulatif

- *Single Instruction Multiple Data* (SIMD)
  - Permet de programmer un cœur CPU ou un/des *Work-item(s)* GPU
  - Plus souvent utilisé pour programmer un cœur CPU
  - Assembleur, fonctions intrinsèques
- *Multi-threads*
  - Permet de programmer plusieurs cœurs CPU
  - Bibliothèques : threads POSIX, OpenMP
- *Multiple Program Multiple Data* (MPMD)
  - Permet de programmer plusieurs cœurs CPU et/ou plusieurs nœuds d'un supercalculateur
  - Bibliothèques : MPI
- *Single Instruction Multiple Threads* (SIMT)
  - Permet de programmer un GPU
  - Langages : OpenCL, CUDA

Table des matières

3 Analyse des codes parallèles

Temps de restitution

Comment comparer deux versions d'un code qui fait la même chose (du point de vue fonctionnel) ?

- Comparer le temps de restitution (ou le temps d'exécution) des deux versions
  - Le programme le plus rapide est le plus efficace
  - Intuitif mais à ne pas perde de vue
- Il faut faire attention à comparer les mêmes temps
  - Erreur classique : comparaison du temps total de l'exécution d'un programme avec juste une sous partie du temps d'exécution d'un autre programme
  - Dans le cas précédent, les deux temps mesurés ne sont pas comparables

Temps de restitution d'un code parallèle

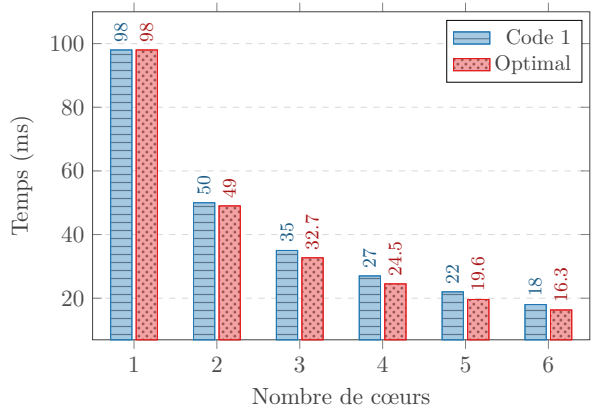
- On considère $\mathcal{D}_1$ (ou $\mathcal{D}_s$) le temps séquentiel (temps sur 1 cœur) d'un code
  - Avec 2 cœurs, on peut espérer diviser au mieux le temps par 2 ($\mathcal{D}_2^m \geq \mathcal{D}_s/2$)
  - Avec 3 cœurs, on peut espérer diviser au mieux le temps par 3 ($\mathcal{D}_3^m \geq \mathcal{D}_s/3$)
- La table suivante présente les temps d'exécution mesurés pour un `Code 1` :

# cœurs ($\mathcal{C}$)	Durée mesurée ($\mathcal{D}^m$)	Durée optimale ($\mathcal{D}^o$)
1	98 ms	98.0 ms
2	50 ms	49.0 ms
3	35 ms	32.7 ms
4	27 ms	24.5 ms
5	22 ms	19.6 ms
6	18 ms	16.3 ms

  - Durée optimale $= \mathcal{D}^o = \mathcal{D}_s/\mathcal{C}$

Visualisation du temps de restitution

- La table précédente n'est pas facile à lire
- Observons les résultats obtenus sur un graphe :



Notion d'accélération (*speedup*)

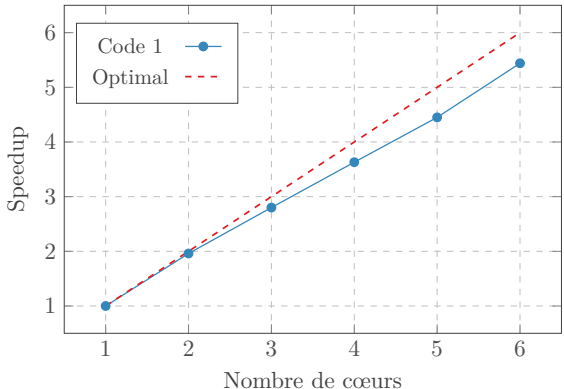
$$\mathcal{S} = \mathcal{D}_s / \mathcal{D}_\mathcal{C},$$

avec $\mathcal{D}_s$ le temps mesuré pour la version 1 cœur du code et $\mathcal{D}_\mathcal{C}$ le temps mesuré pour la version parallèle avec $\mathcal{C}$ cœurs.

# cœurs ($\mathcal{C}$)	Durée ($\mathcal{D}^m$)	Speedup ($\mathcal{S}$)
1	98 ms	1.00
2	50 ms	1.96
3	35 ms	2.80
4	27 ms	3.63
5	22 ms	4.45
6	18 ms	5.44

- Le temps séquentiel est utilisé comme temps de référence

Visualisation de l'accélération



- Visuellement très simple de voir si le `Code 1` est proche de l'optimal
- Une accélération optimale est égale au nombre de cœurs utilisés (pas plus !)

Loi d'Amdahl

- Peut-on indéfiniment augmenter le parallélisme pour accélérer nos codes ?
  - Amdahl a dit non !
  - Pour être plus précis, cela dépend des caractéristiques du code...
  - Si le code est entièrement parallélisable : l'accélération est infinie
  - Si le code n'est PAS entièrement parallélisable : il y a une limite

$$\mathcal{S}_{\max} = \frac{1}{1 - f\mathcal{D}_p},$$

avec $\mathcal{S}_{\max}$ l'accélération (*speedup*) maximale et $f\mathcal{D}_p$ la fraction de temps parallèle dans le code ($0 \leq f\mathcal{D}_p \leq 1$).

Loi d'Amdahl : application numérique

- Prenons un code composé de deux parties :
  - 20 % est intrinsèquement séquentielle
  - 80 % est parallèle
- Quelle est l'accélération maximale que l'on peut atteindre ?

$$\mathcal{S}_{\max} = \frac{1}{1 - f\mathcal{D}_p} = ...$$

Loi d'Amdahl : application numérique

- Prenons un code composé de deux parties :
  - 20 % est intrinsèquement séquentielle
  - 80 % est parallèle
- Quelle est l'accélération maximale que l'on peut atteindre ?

$$\mathcal{S}_{\max} = \frac{1}{1 - f\mathcal{D}_p} = \frac{1}{1 - 0.8} = \frac{1}{0.2} = 5.$$

Loi d'Amdahl : application numérique

- Prenons un code composé de deux parties :
  - 20 % est intrinsèquement séquentielle
  - 80 % est parallèle
- Quelle est l'accélération maximale que l'on peut atteindre ?

$$\mathcal{S}_{\max} = \frac{1}{1 - f\mathcal{D}_p} = \frac{1}{1 - 0.8} = \frac{1}{0.2} = 5.$$

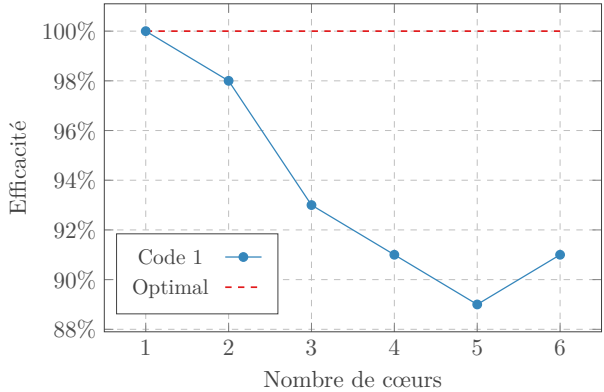
C'est peu si l'on considère des architectures qui ont des dizaines de cœurs CPU !

Notion d'efficacité

- Plusieurs façon de définir l'efficacité $\mathcal{E}$ d'un code
  - À partir du *speedup* : $\mathcal{E} = \mathcal{S}^m / \mathcal{S}^o$
  - À partir du temps de restitution: $\mathcal{E} = \mathcal{D}^m / \mathcal{D}^o$
- L'efficacité est un ratio: $0\% < \mathcal{S} \leq 100\%$
- Pour le `Code 1`, l'efficacité en fonction du nombre de cœurs est la suivante :

# de cœurs ($\mathcal{C}$)	Durée ($\mathcal{D}^m$)	Speedup ($\mathcal{S}$)	Efficacité ($\mathcal{E}$)
1	98 ms	1.00	100%
2	50 ms	1.96	98%
3	35 ms	2.80	93%
4	27 ms	3.63	91%
5	22 ms	4.45	89%
6	18 ms	5.44	91%

Visualisation de l'efficacité



- Équivalent au *speedup*, du moins pour le moment...

Notion de passage à l'échelle (scalabilité)

- La scalabilité d'un code est sa capacité à être efficace quand on augmente le nombre de cœurs de calcul en parallèle
  - On dit qu'un code passe à l'échelle si il est capable de bénéficier de la puissance de plusieurs cœurs
- Comment mesurer la scalabilité d'un code ? Comment savoir si un code ne passe pas à l'échelle ?
  - Pas de réponse simple...
- Il existe deux modèles très utilisés pour caractériser la scalabilité d'un code parallèle :
  - La scalabilité dite "forte" (*strong scalability*)
  - La scalabilité dite "faible" (*weak scalability*)

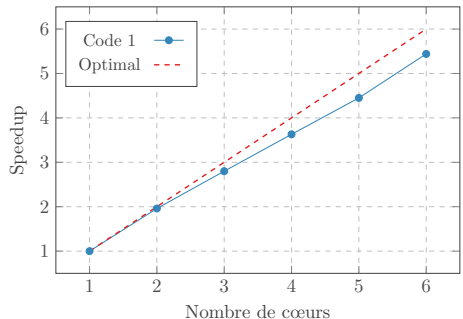
Scalabilité forte du `Code 1`

- Mesure le temps de restitution en fonction du nombre de cœurs
- Avec **une taille de problème constante**
- Par exemple pour le `Code 1`, avec un problème de taille 100 :

# de cœurs	Taille	Durée	Speedup
1	100	98 ms	1.00
2	100	50 ms	1.96
3	100	35 ms	2.80
4	100	27 ms	3.63
5	100	22 ms	4.45
6	100	18 ms	5.44

Scalabilité forte du `Code 1` : visualisation

On observe généralement la scalabilité forte sur un graphe d'accélération (comme vu précédemment).



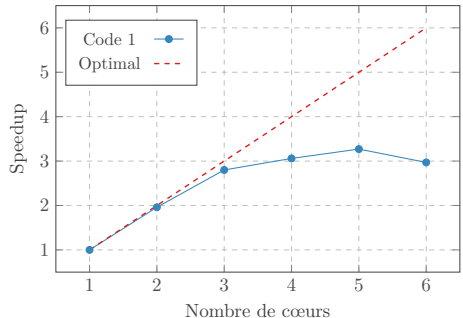
Ici pour 6 cœurs, le `Code 1` obtient une accélération de 5.4, on peut en conclure que ce code passe bien à l'échelle jusqu'à 6 cœurs.

Passage à l'échelle du `Code 2`

- Considérons un nouveau `Code 2`
- Voici les mesures de ce code :

# de cœurs	Taille	Durée	Speedup
1	100	98 ms	1.00
2	100	50 ms	1.96
3	100	35 ms	2.80
4	100	32 ms	3.06
5	100	30 ms	3.27
6	100	33 ms	2.97

Scalabilité forte du `Code 2`



- On peut voir que la scalabilité forte du `Code 2` est mauvaise
- Il est courant qu'à partir d'un certain nombre de cœurs, le parallélisme ne permette plus d'accélérer le code :-(

Scalabilité faible

- Ce modèle considère le temps de restitution en fonctions du nombre de cœurs
- **Et la taille du problème augmente proportionnellement au nombre de cœurs !**
- Calculer le *speedup* n'a pas de sens si la taille du problème n'est pas constante
- MAIS on peut calculer une efficacité : $\mathcal{E} = \mathcal{D}^s/\mathcal{D}^m$

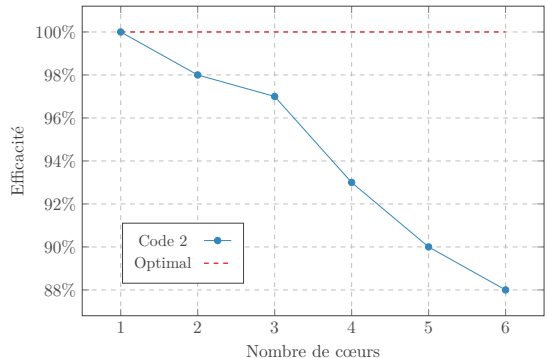
Intuition : si on ne peut pas calculer plus vite un problème à une taille donnée, peut-on calculer en un même temps un problème plus gros ?

Bien souvent oui, et cela est plus facile ! C'est ce que l'on fait la plupart du temps en calcul haute performance : les modèles scientifiques sont de plus en plus raffinés = la taille du problème augmente.

Scalabilité faible du `Code 2`

Mesures pour le `Code 2` :

# de cœurs	Taille	Durée	Efficacité
1	100	098 ms	100%
2	200	100 ms	98%
3	300	101 ms	97%
4	400	105 ms	93%
5	500	109 ms	90%
6	600	111 ms	88%

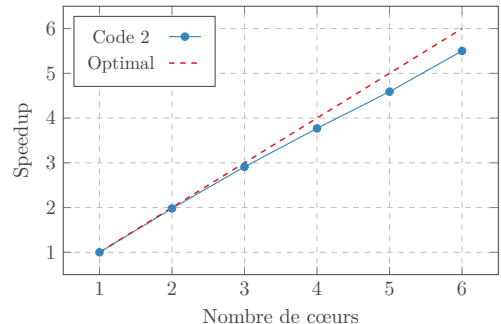


- La scalabilité faible du `Code 2` est bonne ($\approx 90\%$ pour 6 cœurs)
- Pourquoi la scalabilité forte est mauvaise ?
  - — Loi d'Amdahl : pas assez de parallélisme pour une petite taille de problème

Scalabilité forte du `Code 2` : AGAIN !

- Nouvelle taille de problème : 600

# de cœurs	Taille	Durée	Speedup
1	600	611 ms	1.00
2	600	308 ms	1.98
3	600	210 ms	2.91
4	600	162 ms	3.77
5	600	133 ms	4.59
6	600	111 ms	5.50



- Pour une taille de problème plus grosse, la scalabilité forte est bonne
- Pas toujours possible de faire des tests de scalabilité forte : manque de temps
- Pas toujours possible de faire des tests de scalabilité faible : impossible d'avoir des tailles de problème toujours plus grande

Table des matières

Nombre d'opérations arithmétiques

- Le nombre d'opérations arithmétiques d'un code est une caractéristique importante
- Exemple du nombre d'opérations flottantes (*flops*) dans un noyau de calcul qui fait une somme :

```
1 float sum(float *values, int n) {
2     float sum = 0.f;
3
4     // total flops = n * 1
5     for(int i = 0; i < n; i++)
6         sum = sum + values[i]; // 1 flop à cause de l'addition
7
8     return sum;
9 }
```

Nombre d'opérations par seconde

- Métrique très utilisée dans le calcul haute performance et plus particulièrement le nombre d'opérations **flottantes** par seconde (flop/s)
- Le ratio flop/s peut être directement comparé avec la performance crête d'une architecture de calcul
- Un critère de la bonne utilisation (ou pas) de l'architecture matérielle

Performance crête d'un processeur

- C'est la capacité de calcul maximale du processeur
- On peut la calculer à partir de notre connaissance de l'architecture matérielle :

$$peakPerf = nOps \times freq \times nCores,$$

avec $nOps$ le nombre d'opérations flottantes que l'architecture peut calculer en un cycle, $freq$ la fréquence du processeur et $nCores$ le nombre de cœurs du processeur.

Performance crête d'un processeur : exemple

4 Analyse de la performance d'un *kernel*

CPU name	Core i7-2630QM
Architecture	Sandy Bridge
Vect. inst.	AVX-256 bit (4 double, 8 simple)
Frequency	2 GHz
Nb. cores	4

Table: Spécifications depuis http://ark.intel.com/products/52219

La performance crête en simple précision est :

$$peakPerf_{sp} = nOps \times freq \times nCores = (2 \times 8) \times 2 \times 4 = 128\ Gflop/s$$

La performance crête en double précision est :

$$peakPerf_{dp} = nOps \times freq \times nCores = (2 \times 4) \times 2 \times 4 = 64\ Gflop/s$$

- $nOps = 2 \times vectorSize$ parce que l'architecture permet de calculer 2 instructions par cycle (`vadd` et `vmul`)

Intensité arithmétique

- Parfois (voire souvent) les Gflop/s mesurés sont loin de la performance crête
  - Le code est mal optimisé
  - Il n'est pas possible d'atteindre la performance crête
  - Dans la plupart des cas, les deux sont vrais
- Avec l'intensité arithmétique nous prenons en considération les accès mémoire :

$$AI = \frac{flops}{memops}.$$

Intensité arithmétique: exemple

4 Analyse de la performance d'un *kernel*

```
1  float sum(float *values, int n) {
2      float sum = 0.f; // on ne compte pas les accès à sum comme des accès à la mémoire
3                       // car cette variable sera optimisée en registre
4
5      // total flops = n * 1  ||  total memops = n * 1
6      for(int i = 0; i < n; i++)
7          sum = sum + values[i]; // 1 flop à cause d'1 addition, 1 memop à cause
8                                 // d'1 accès dans la tabelau `values`
9
10     return sum;
11 }
```

- L'intensité arithmétique de `sum` est : $AI_{\texttt{sum}} = \frac{n \times 1}{n \times 1} = 1$
- Plus l'intensité arithmétique est élevée plus le code est limité par les unités de calcul
- Plus l'intensité arithmétique est faible plus le code est limité par les accès mémoire

Intensité opérationnelle

- Par rapport à l'intensité arithmétique, l'intensité opérationnelle tient compte de la taille des données en mémoire :

$$OI = \frac{flops}{memops \times sizeOfData} = \frac{AI}{sizeOfData},$$

$sizeOfData$ dépend du type des données du code, `int` et `float` occupent 4 octets, `long long int` et `double` occupent 8 octets.

- Dans le code précédent (`sum`), les accès mémoire sont sur des `float` et donc l'intensité opérationnelle vaut : $OI_{\texttt{sum}} = \frac{n \times 1}{(n \times 1) \times 4} = \frac{1}{4}$

Intensité opérationnelle : exemple

4 Analyse de la performance d'un *kernel*

Somme en simple précision :

```
1 // AI = 1  ||  OI = 1/4
2 float sum1(float *values, int n) {
3     float sum = 0.f;
4     for(int i = 0; i < n; i++)
5         sum = sum + values[i];
6     return sum;
7 }
```

Somme en double précision :

```
1 // AI = 1  ||  OI = 1/8
2 double sum2(double *values, int n) {
3     double sum = 0.0;
4     for(int i = 0; i < n; i++)
5         sum = sum + values[i];
6     return sum;
7 }
```

- Les noyaux de calcul `sum1` et `sum2` ont la même intensité arithmétique
- L'intensité opérationnelle de `sum2` est plus élevée que celle de `sum1` :
  - Le noyau de calcul `sum2` est plus limité par les accès mémoire que le noyau de calcul `sum1`

Le modèle *Roofline*

- Un modèle qui permet de borner la performance maximale atteignable
- Il prend en considération :
  - La bande passante mémoire (RAM)
  - La performance crête du processeur
- En fonction de l'intensité opérationnelle, le code est limité soit par la bande passante mémoire soit par la performance crête du processeur

$$Attainable\ Gflop/s = min \begin{cases} Peak\ floating\ point\ performance, \\ Peak\ memory\ bandwidth \times OI. \end{cases}$$

Mesure de la bande passante mémoire

- La bande passante mémoire ou le débit mémoire représente le nombre d'octets qui peuvent être lus/écrits depuis la RAM en une seconde (o/s ou Go/s)
- Comment connaître la bande passante mémoire
  - Il est possible de calculer une valeur théorique
  - Mais on préfère souvent utiliser un programme micro-benchmark : STREAM
- STREAM est un petit code relativement simple pour mesurer la bande passante mémoire
  - Donne des résultats rapidement et fiables

Le modèle *Roofline*: example

4 Analyse de la performance d'un *kernel*

Reprenons le même processeur que précédemment :

CPU name	Core i7-2630QM
Peak perf sp	128 GFlop/s
Peak perf dp	64 GFlop/s
Mem. bandwidth	17.6 GB/s

Somme en simple précision :

```
1 // AI = 1  ||  OI = 1/4
2 float sum1(float *values, int n) {
3     float sum = 0.f;
4     for(int i = 0; i < n; i++)
5         sum = sum + values[i];
6     return sum;
7 }
```

Somme en double précision :

```
1 // AI = 1  ||  OI = 1/8
2 double sum2(double *values, int n) {
3     double sum = 0.0;
4     for(int i = 0; i < n; i++)
5         sum = sum + values[i];
6     return sum;
7 }
```

Le modèle *Roofline*: exemple

Peak perf sp	128 GFlop/s
Peak perf dp	64 GFlop/s
Mem. bandwidth	17.6 GB/s

Pour `sum1`, l'intensité opérationnelle est : $AI_{sum1} = 1/4$.
Appliquons le modèle *Roofline* :

$$Attainable\ Gflop/s = min \begin{cases} Peak\ floating\ point\ performance, \\ Peak\ memory\ bandwidth \times OI. \end{cases}$$

$$\Rightarrow$$

$$Attainable\ Gflop/s_{\texttt{sum1}} = min \begin{cases} 128\ Gflop/s, \\ 17.6 \times \frac{1}{4}\ Gflop/s. \end{cases} = 4.4\ Gflop/s$$

Le modèle *Roofline*: exemple

4 Analyse de la performance d'un *kernel*

Peak perf sp	128 GFlop/s
Peak perf dp	64 GFlop/s
Mem. bandwidth	17.6 GB/s

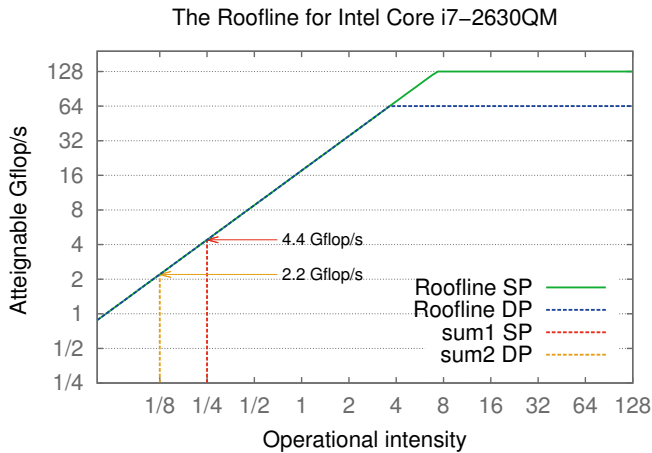
Pour `sum1`, l'intensité opérationnelle est : $AI_{sum2} = 1/8$.
Appliquons le modèle *Roofline* :

$$Attainable\ Gflop/s = min \begin{cases} Peak\ floating\ point\ performance, \\ Peak\ memory\ bandwidth \times OI. \end{cases}$$

$$\Rightarrow$$

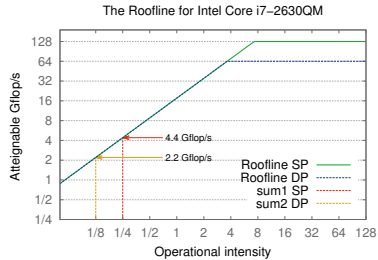
$$Attainable\ Gflop/s_{\texttt{sum2}} = min \begin{cases} 64\ Gflop/s, \\ 17.6 \times \frac{1}{8}\ Gflop/s. \end{cases} = 2.2\ Gflop/s$$

Le modèle *Roofline*: visualisation



Le modèle *Roofline*: visualisation

4 Analyse de la performance d'un *kernel*



- Il y a deux *Rooflines* différentes
  - — Une pour les calculs en simple précision
  - — Une pour les calculs en double précision
- Ici il apparaît clairement que `sum1` et `sum2` sont limités par la bande passante mémoire

Q&R

Merci pour votre écoute !
Avez-vous des questions ?