Machine/Deep Learning is High Performance Computing and, finally, parallel computing

27.06.2018

Bogdan Vulpescu
LPC+

Sources

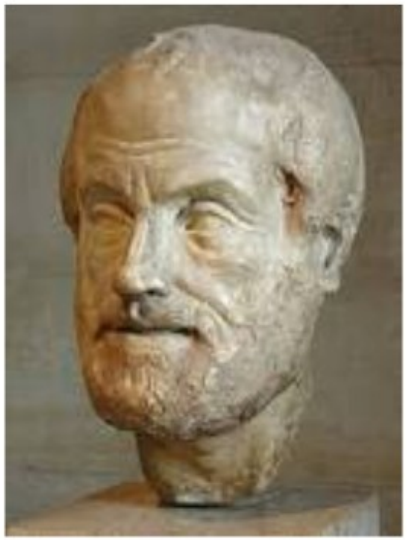
[1] Vincent Barra, ISIMA/LIMOS,
Recontres AuDACES, 7 juin 2018
http://audaces.asso.st/index.php?calcul=rencontre-2018

[2] Christophe Tilmant, IBP, 1er Café AuDACES, 23 avril 2018

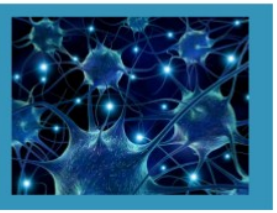
[3] Jonathan Rouzaud-Cornabas, LIRIS, INRIA, Journées GPGPU, 12-13 juin 2018

LIRIS = Laboratoire d'InfoRmatique en Image et Systèmes d'information
INRIA = Institute National de Recherche en Informatique et en Automatique

- 300 avant JC : Aristote : la pensée est une suite d'éléments conceptuels associés.
- 1873 : Alexander Bain : groupements neuronaux
- 1943 : Mc Culloch et Pitts : modèle MCP : précurseur des réseaux de neurones
- 1949 : Donald Hebb : père des réseaux de neurones, règle d'apprentissage de Hebb
- 1958 : Franck Rosenblatt : perceptron
- 1974 : Paul Werbos : rétropropagation
- 1980 : Teuvo Kohonen : SOM
- 1980 : Kunihiko Fukushima : Neocogitron, l'inspirateur des CNN
- 1985 : Geoffrey Hinton : Machines de Boltzman
- 1986 : Michael Jordan : réseaux récurrents
- 1990 : Yan LeCun : réseaux convolutifs
- 1997 : Hochreiter et Schmidhuber : LSTM
- 2006 : Geoffrey Hinton : Deep Belief Network
- 2012 : ImageNet
- 2015 : resnet : réseau profond à 152 couches



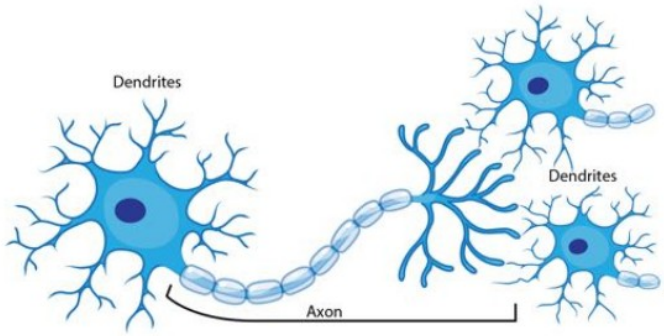
Principe des réseaux de neurones

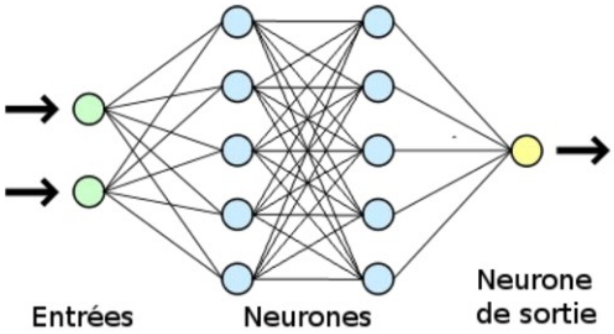


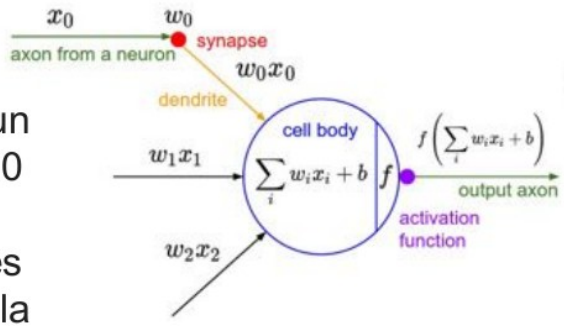
Les réseaux neuronaux sont construits sur un paradigme biologique, celui du neurone formel.

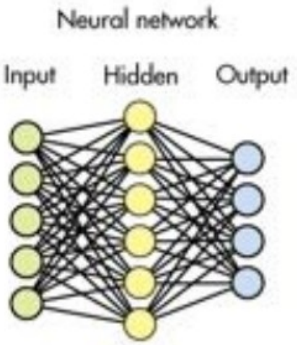
- **1940**: Réseaux de neurones artificiels;
- **1950**: Perceptron;
- **1957**: Premier réseau simple;
- **1986**: Perceptrons multicouches, parallèlement, par David Rumelhart et Yann LeCun;
- **1980**: Rétropropagation du gradient.

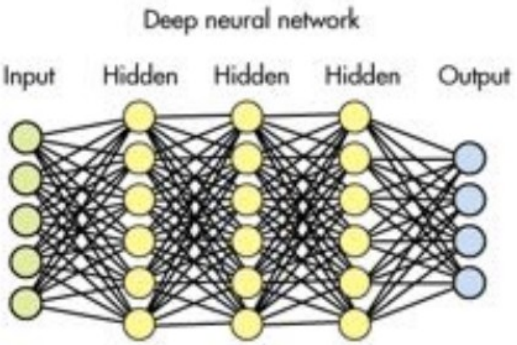
L'apprentissage profond connait un essor inédit dans les années 2010 avec l'émergence de la disponibilité de données massives (« *big data* ») et l'accélération de la vitesse de calcul des processeurs.



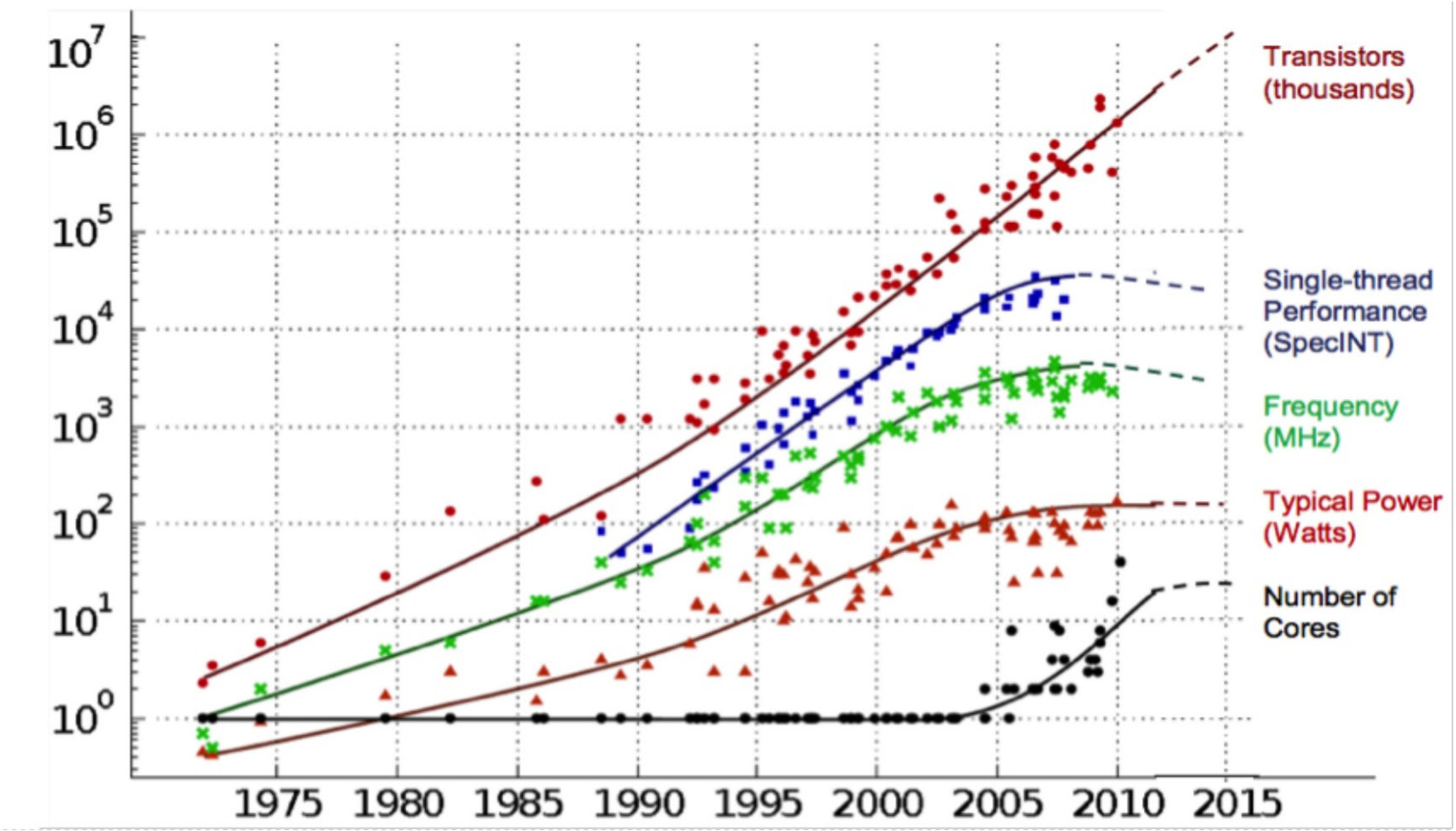








Loi de Moore : La réalité



Loi de Moore et Parallélisme: Conclusion

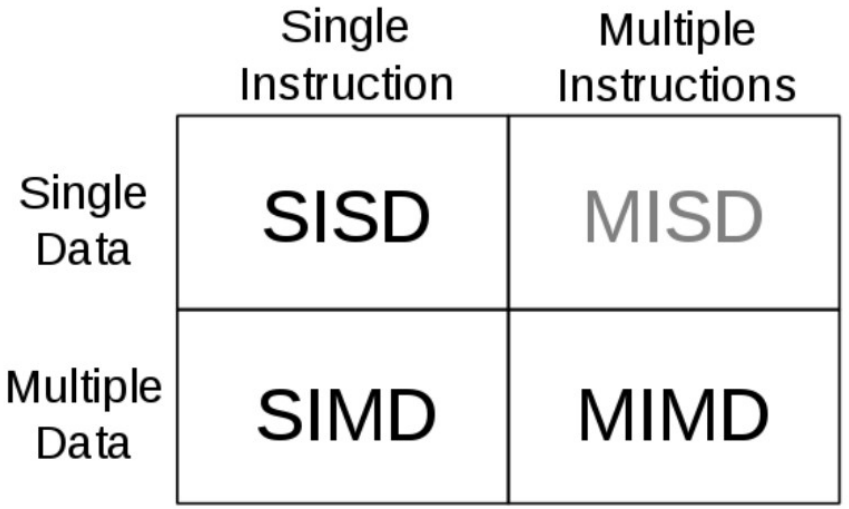
- La Loi de Moore tient toujours mais pas comme on s'y attendait
- Le parallélisme est obligatoire et présent partout
- Le chargement des données est aussi important que la rapidité de calcul
- Même problème sur les grands systèmes parallèles
  - Clusters
  - Super-calculateurs
  - Clouds
  - ...

Qu'est-ce que le calcul parallèle ?

- Pouvoir accélérer une application en
  1. Divisant cette applications en sous-tâches
  2. Exécuter ces sous-tâches en parallèles sur des unités différentes

- Pour réussir, il faut être capable de
  1. Trouver le parallélisme dans l'application
  2. Trouver le bon grain de calcul/échange de données
  3. Avoir des connaissances pour concevoir une solution efficace sur la machine cible

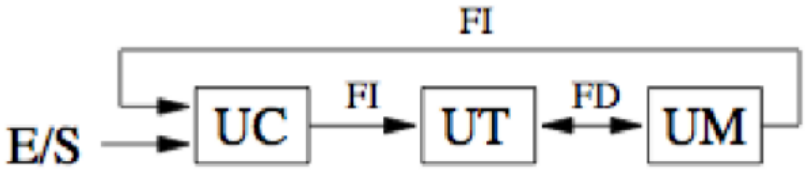
Classification de Flynn



- Caractérise les machines suivant leurs flots de données et d'instructions

SISD: Single Instruction, Single Data stream

- Machines séquentielles “classiques“
- Chaque opération s'effectue sur une donnée à la fois

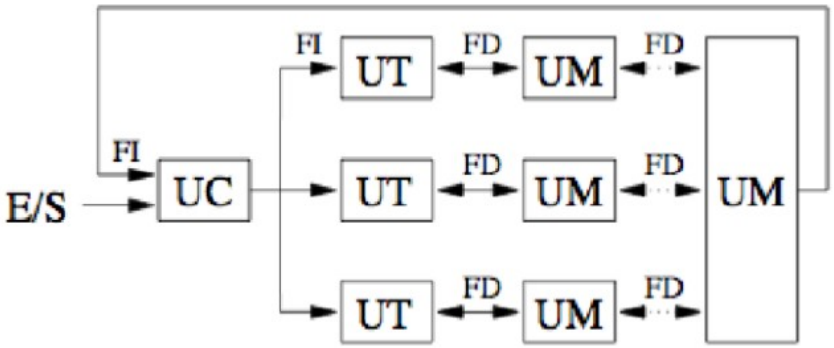


- UC = Unité de Contrôle (chargée du séquencage des instructions)
- UT = Unité de traitement (effectue les opérations)
- FI = Flot d'Instructions
- UM = Unité Mémoire (contient les instructions et les données)
- FD = Flot de Données

- Modèle de Von Neuman (1945)

SIMD: Single Instruction stream, Multiple Data stream

- Unités de calcul totalement synchronisées
- Exécution conditionnelle avec flag de masquage



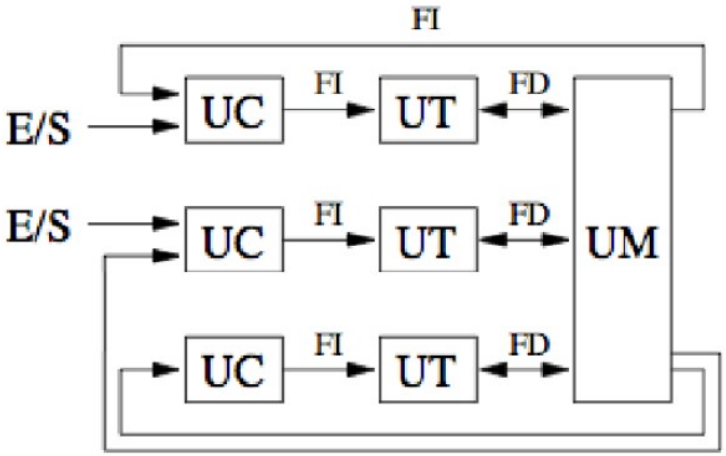
- Machines adaptées aux traitements très réguliers (opérations matricielles, FFT, imagerie).
- Pas adapté du tout aux opérations irrégulières.

MIMD: Multiple Instructions stream, multiple data stream

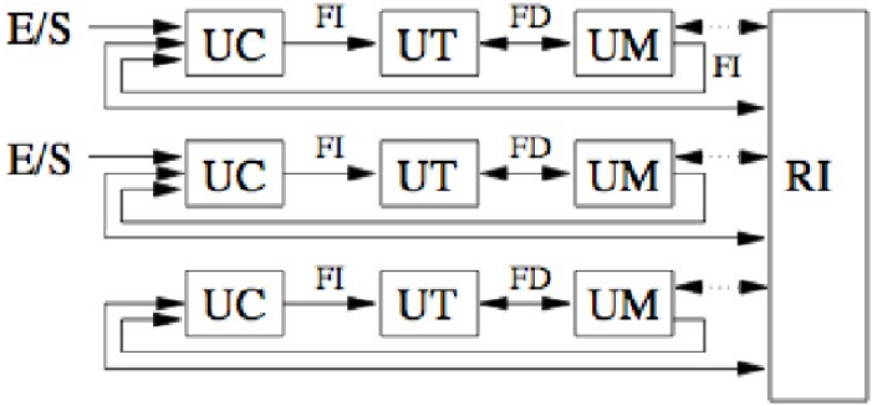
- Machines multi-processeurs
- Chaque processeur exécute son propre code de manière asynchrone et indépendante

Deux sous-classes

Mémoire partagée



Mémoire distribuée

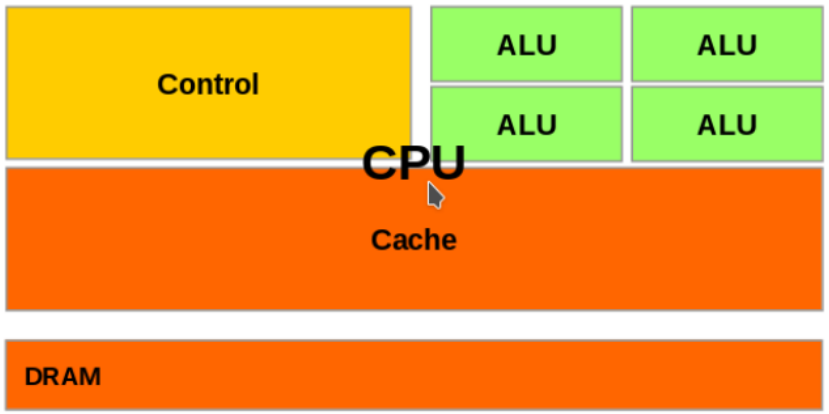


A mi-chemin entre SIMD et MIMD: SPMD (*Single Program, Multiple Data*)

RI = réseau d'interconnexion

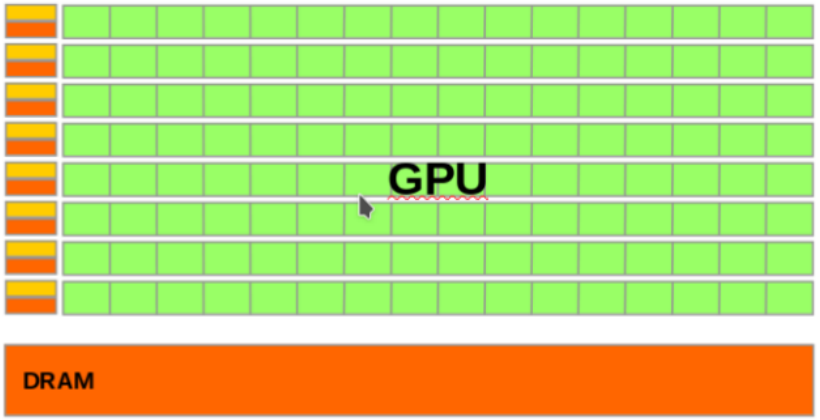
CPU: Conception orienté latence

- Haute fréquence d'horloge
- Caches de grandes tailles
  - Converti les accés à haute latence dans la mémoire en accés à basse latence dans le cache
- Système de contrôle sophistiqué
  - Prédiction de branche pour réduire la latence dû aux branches
  - Chargement de données pour réduire la latence dû à l'accés aux données
- Puissante unité arithmétique et logique (ALU)
  - Réduction de la latence des opérations



GPU: Conception orienté débit

- Fréquence d'horloge modérée
- Caches de petites tailles
  - Pour maximiser le débit mémoire
- Contrôle simple
  - Pas de prédiction de branches
  - Pas de chargement de données
- Unite arithmétique et logique (ALU) à faible consommation
  - Nombreuses, à haute latence mais fortement pipeliner pour de fort débit
- Nécessite un très grand nombre de threads pour que la latence soit tolérable

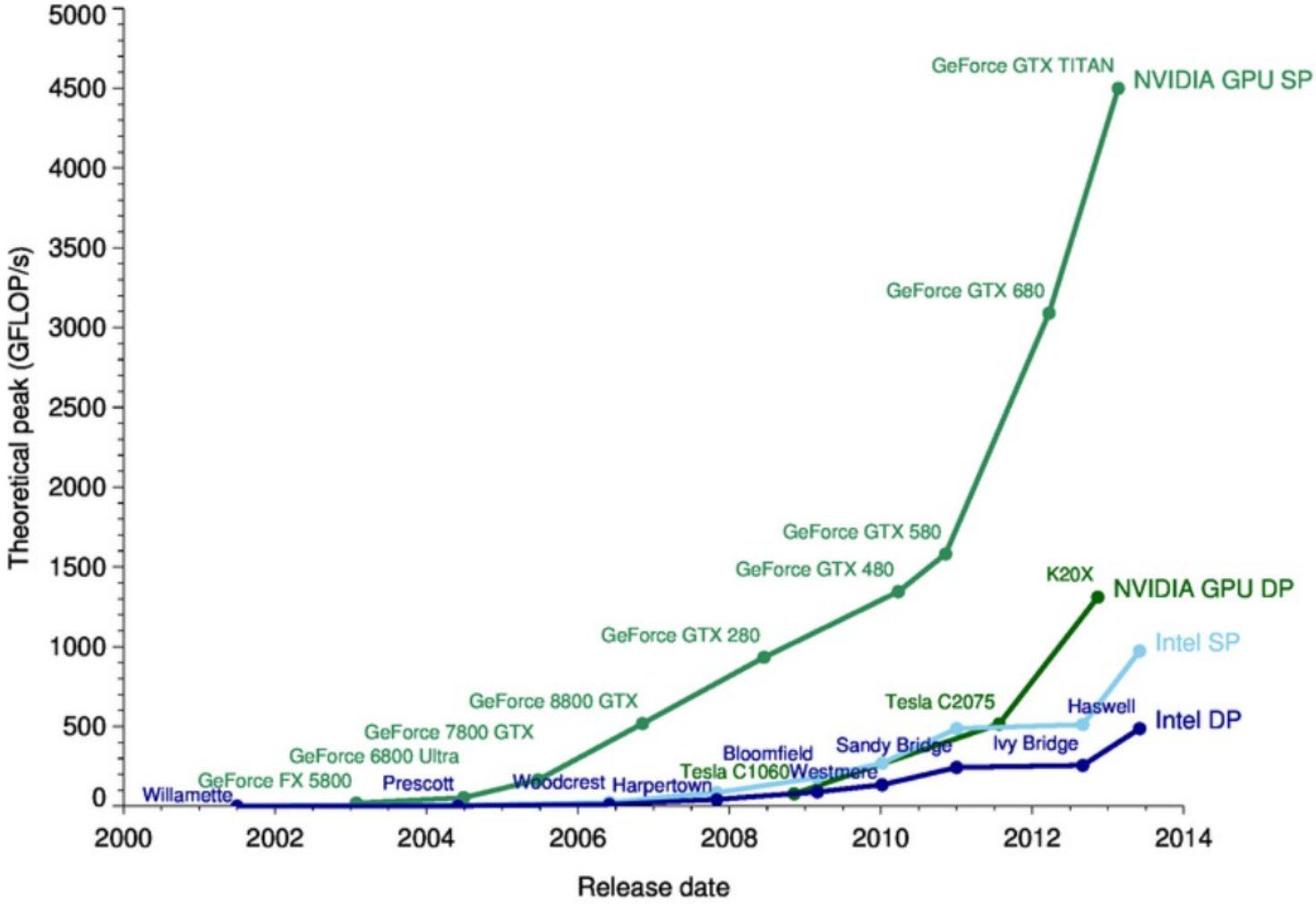


La stratégie gagnante est d'utiliser les deux

- CPUs pour les parties séquentielles où la latence est critique
  - CPUs peuvent être 10+X plus rapide que les GPUs pour le code séquentiel
- GPUs pour les parties parallèles où le débit est critique
  - GPUs peuvent être 10+X plus rapide que les CPUs pour le code parallèle

[3]

Evolution des performances CPU/GPU



SP/DP (Single/Double Precision)

CPU optimization

1

```
//_____________________________________
int main()
{
  int n = 1000000;
  for (int k = 1; k <= 256; k <<= 1) {
    f(n*k);
  }
  return 0;
}

//_____________________________________
void f(int n)
{
  for (int i = 0; i < n; ++i) {
  }
}
```

```
g++ -o a.out main.cxx

K =      1   Real time =   0.005616
K =      2   Real time =   0.009323
K =      4   Real time =   0.015542
K =      8   Real time =   0.027990
K =     16   Real time =   0.046137
K =     32   Real time =   0.075879
K =     64   Real time =   0.150435
K =    128   Real time =   0.301202
K =    256   Real time =   0.602216

g++ -O -o a.out main.cxx

K =      1   Real time =   0.000001
K =      2   Real time =   0.000001
K =      4   Real time =   0.000001
K =      8   Real time =   0.000001
K =     16   Real time =   0.000001
K =     32   Real time =   0.000001
K =     64   Real time =   0.000001
K =    128   Real time =   0.000001
K =    256   Real time =   0.000001
```

CPU optimization

2

```
//_____________________________________
int main()
{
  int n = 1000000;
  for (int k = 1; k <= 256; k <<= 1) {
    f(n*k);
  }
  return 0;
}

//_____________________________________
void f(int n)
{
  int jloc = 0;
  for (int i = 0; i < n; ++i) {
    ++jloc;
  }
}
```

```
g++ -o a.out main.cxx

K =      1   Real time =   0.004732
K =      2   Real time =   0.008863
K =      4   Real time =   0.015801
K =      8   Real time =   0.028363
K =     16   Real time =   0.046283
K =     32   Real time =   0.072167
K =     64   Real time =   0.128914
K =    128   Real time =   0.257803
K =    256   Real time =   0.516108

g++ -O -o a.out main.cxx

K =      1   Real time =   0.000001
K =      2   Real time =   0.000001
K =      4   Real time =   0.000001
K =      8   Real time =   0.000001
K =     16   Real time =   0.000001
K =     32   Real time =   0.000001
K =     64   Real time =   0.000001
K =    128   Real time =   0.000001
K =    256   Real time =   0.000001
```

CPU optimization

3

```
int jglo = 0;
//_____________________________________
int main()
{
  int n = 1000000;
  for (int k = 1; k <= 256; k <<= 1) {
    f(n*k);
  }
  return 0;
}

//_____________________________________
void f(int n)
{
  for (int i = 0; i < n; ++i) {
    ++jglo;
  }
}
```

```
g++ -o a.out main.cxx

K =      1   Real time =   0.004980
K =      2   Real time =   0.008814
K =      4   Real time =   0.015661
K =      8   Real time =   0.027715
K =     16   Real time =   0.046168
K =     32   Real time =   0.072063
K =     64   Real time =   0.128899
K =    128   Real time =   0.257820
K =    256   Real time =   0.515760

g++ -O -o a.out main.cxx

K =      1   Real time =   0.000820
K =      2   Real time =   0.001666
K =      4   Real time =   0.003303
K =      8   Real time =   0.005827
K =     16   Real time =   0.011463
K =     32   Real time =   0.020162
K =     64   Real time =   0.034262
K =    128   Real time =   0.055564
K =    256   Real time =   0.088412
```

PID with Feed-Forward Neural Network (1)

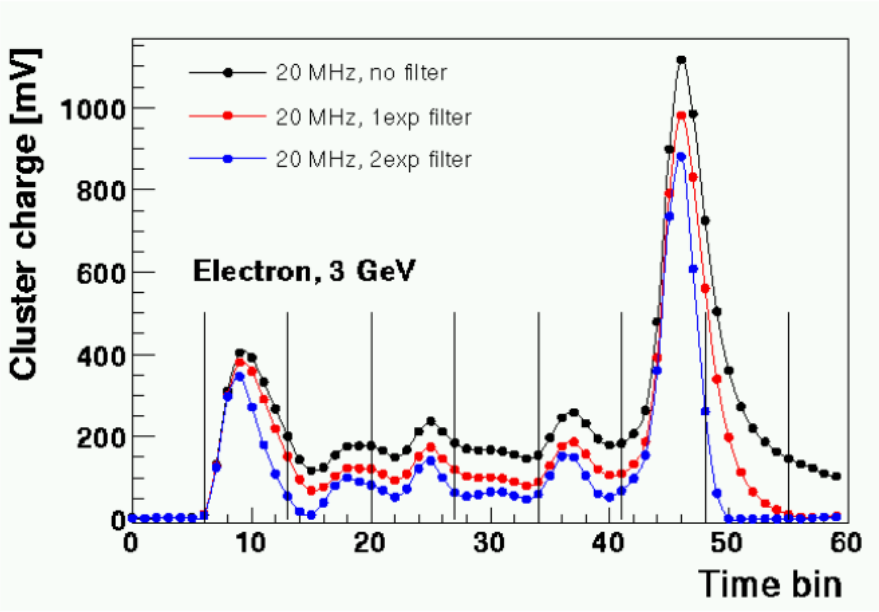
JetNet 3.0 (Lund)

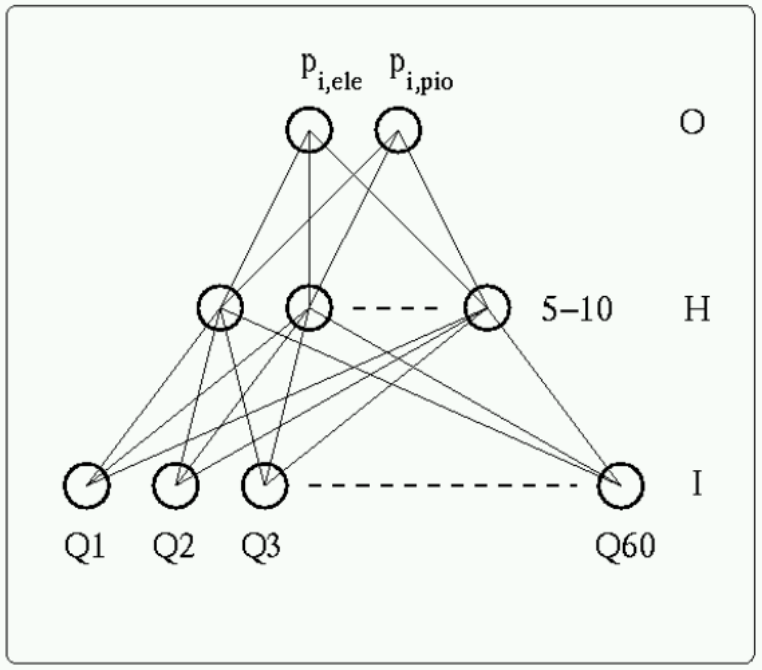
$$Q_i = \sum_{1}^{n} q_i^j \qquad i = 1...6 \qquad n = 20, 40, 60$$

$P_{i,ele}$ and $P_{i,pio}$ used for likelihood L(6) or a second NN.

Correlations between the samples!

- Short drift time
- Electronic response
- Ion tails in the drift signal!

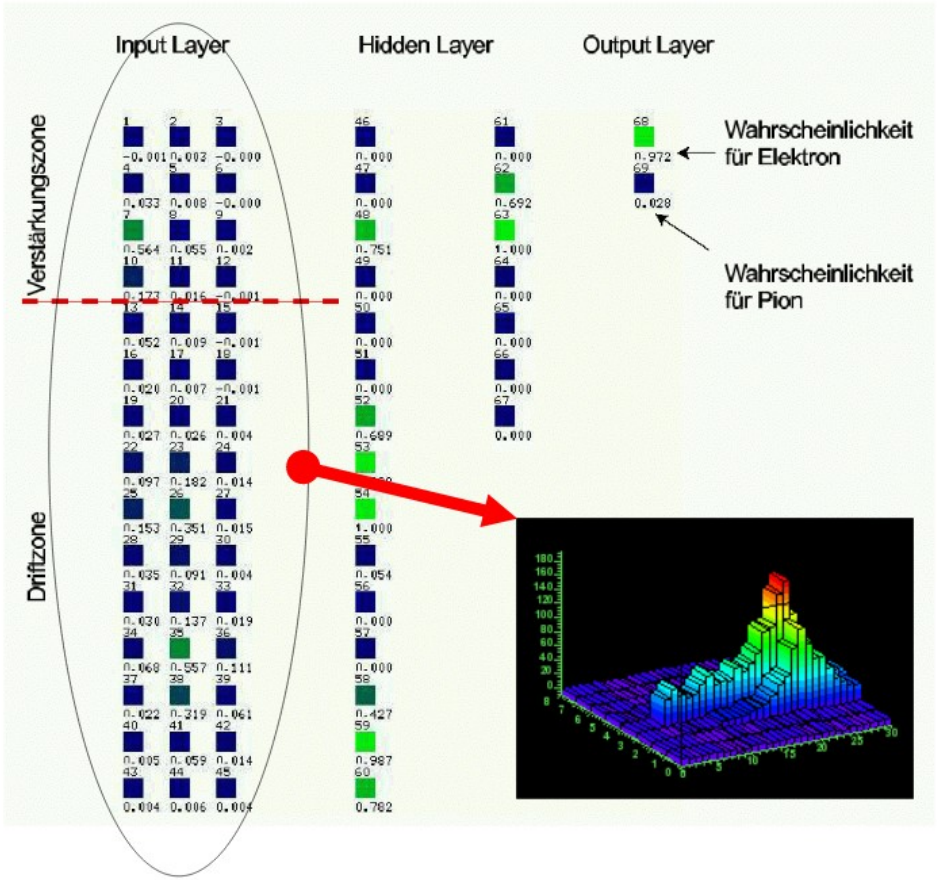


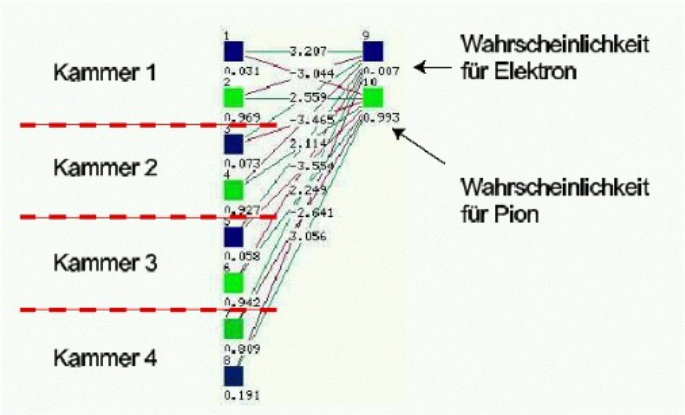


PID with Feed-Forward Neural Network (2)

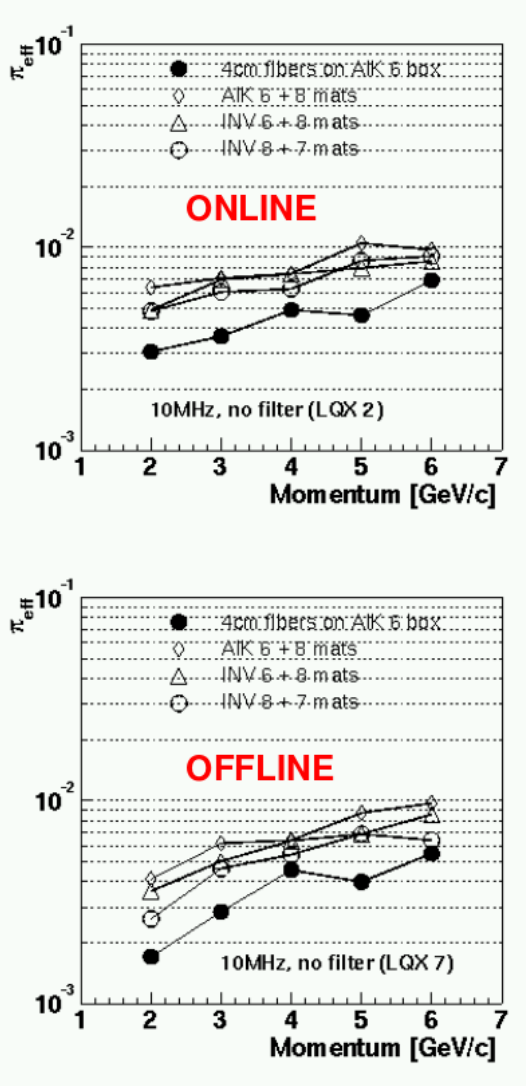
The Stuttgart neural Network Simulator (SNNS) - A. Wilk, Uni. Muenster, Diploma Thesis

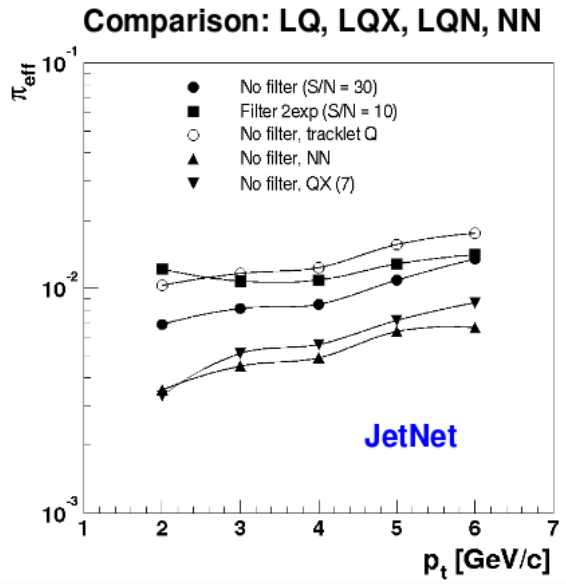
Use full cluster information (3-pads amplitude) in time direction.

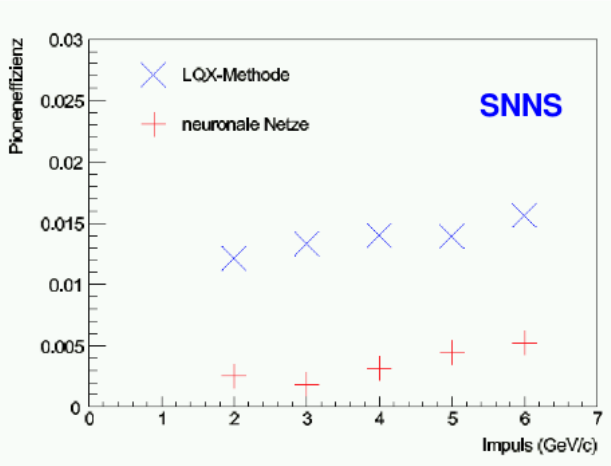




Results - momentum scan

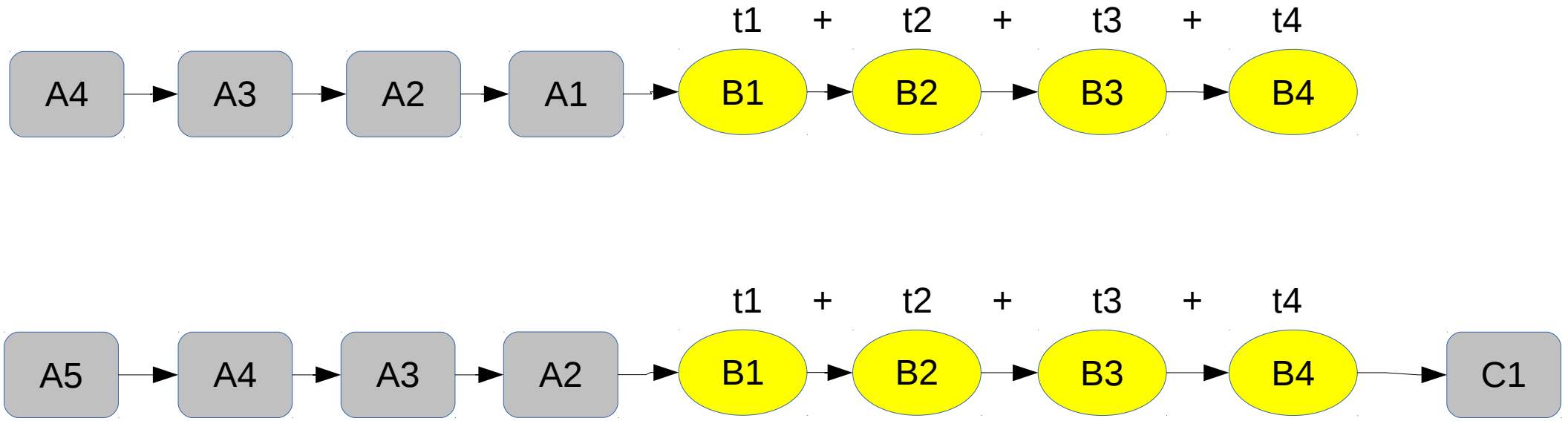






Latency versus throughput

latency



Latency versus throughput

throughput