Impact de l'utilisation de la vectorisation et du multithreading sur la performance et la consommation énergétique

Étude sur les cartes de développement Nvidia Jetson

Sylvain Jubertie, Emmanuel Melin, Naly Raliravaka
emmanuel.melin@univ-orleans.fr

LIFO, Université d'Orléans

Cascimodot 2022
20/06/2022

Optimisations de la performance et/ou de l'énergie ?

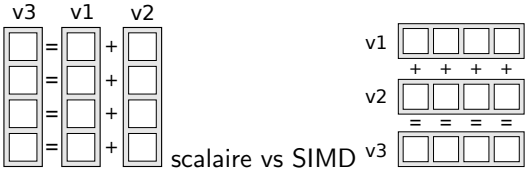
Aujourd'hui presque toutes les architectures (superordinateurs/smartphones) contiennent des processeurs avec:

- Multiples cœurs
- Chaque cœur contient une unité SIMD (Single Instruction on Multiple Data)

Scénarios possibles:

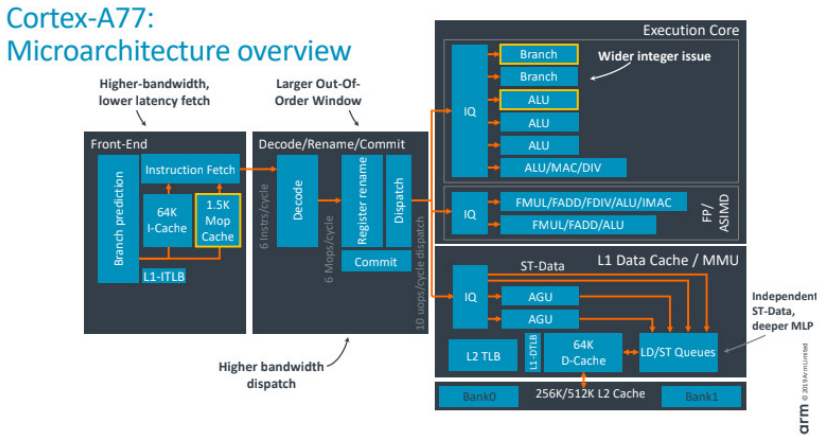
- Maximiser les performances
- Maximiser l'efficacité énergétique (embarqué, contraintes économiques)
- Ajuster à une contrainte de performance (temps réel)
- Respecter une enveloppe de puissance / contrainte thermique

Unités SIMD



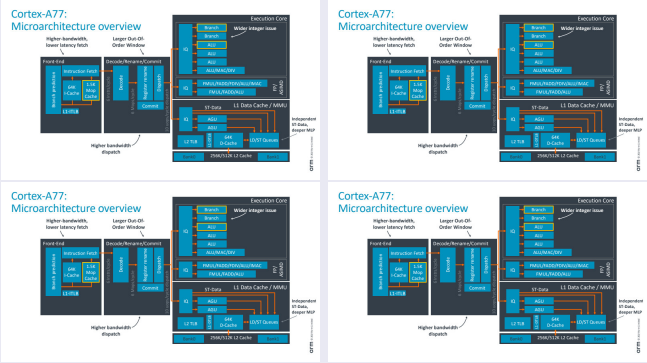
scalaire vs SIMD

- Appliquer la même instruction sur des données multiples (vecteur de données)
- Variété de technologies (en longueur et nombre de vecteurs):
  - x86 (Intel/AMD): SSE(128-bit), AVX(256-bit), AVX-512(512-bit)
  - Arm: NEON/ASIMD(128-bit), SVE(jusqu'à 2048-bit)
  - Power (IBM): VMX, VSX(128-bit)
  - NEC Aurora Tsubasa
  - GPUs



Arm Cortex-A77: 2 unités ASIMD(128-bit), jusqu'à 2 SP FMAs/cycle (SP FMA: Single precision Fused Multiply-Add)

Processeur de la Jetson



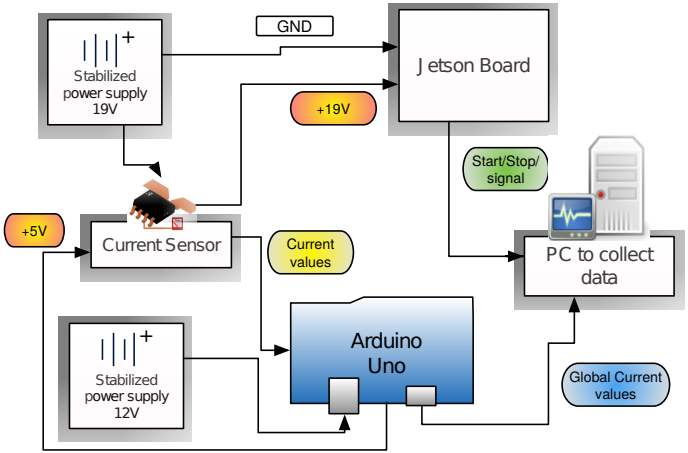
Jetson TX1 Quad-Core ARM Cortex-A57: jusqu'à 8 SP FMAs/cycle

Impact sur la puissance / consommation d'énergie

Utilisation de DVFS (Dynamic Voltage and Frequency Scaling) pour:

- Mesurer l'impact de la mémoire et des fréquences CPU sur la performance / consommation d'énergie.
- Mesurer l'impact de l'utilisation d'unités SIMD, de plusieurs cœurs ou les deux sur les performances et l'énergie: l'utilisation d'unités SIMD et de plusieurs cœurs peut nécessiter plus de puissance mais pour un temps plus court.
- Considérer différents codes: l'impact peut dépendre du scénario: Memory Bound vs Compute Bound
- Comparer un seul cœur avec SIMD vs 4 cœurs avec des unités scalaires.

Plateforme expérimentale



Spécifications de la carte Jetson

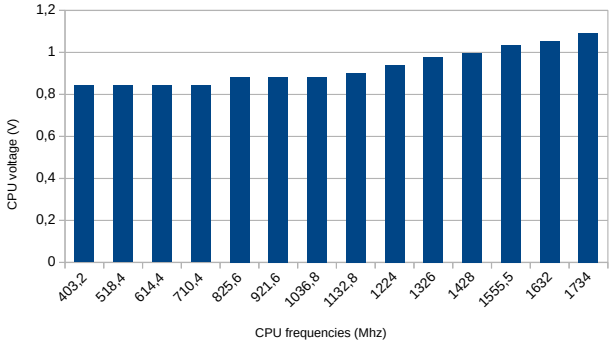
	Jetson TK1	**Jetson TX1**
processor	4x Cortex-A15 (32-bit)	4x Cortex-A57 (64-bit)
cpu freq.	51 - 2,065	102 - 1,734
cache	32KB L1D/128KB L2	32KB L1D/2MB L2
memory	2GB DDR3L	4GB LPDDR4
mem. freq.	12.75 - 924.0	40.8 - 1,600.0
GPU	192 Kepler cores	256 Maxwell cores
GPU freq.	72.0 - 852.0	76.8 - 998.4

Formule de consommation d'énergie

$$P_{avg} = fCV^2 + P_{static}$$

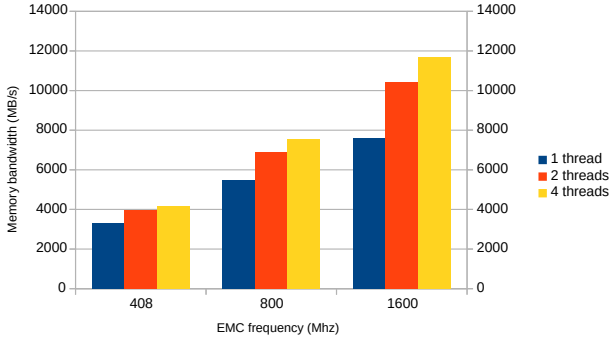
- P_{avg}: puissance consommée moyenne
- f: fréquence CPU
- C: capacité des transistors
- V: tension d'alimentation
- P_{static}: puissance consommée par le CPU au repos

tension du CPU



- L'augmentation de la fréquence du CPU nécessite d'augmenter la tension du CPU
- La tension du processeur augmente non linéairement
- De 1.428GHz à 1.734GHz: f x1.21, V x1.1 $\rightarrow$ fV^2 x1.46

Bande passante mémoire (STREAM)



- La bande passante mémoire dépend de la fréquence et du nombre de cœurs
- La bande passante n'augmente pas linéairement avec la fréquence
- Besoin d'utiliser 4 threads pour profiter pleinement de la bande passante de mémoire

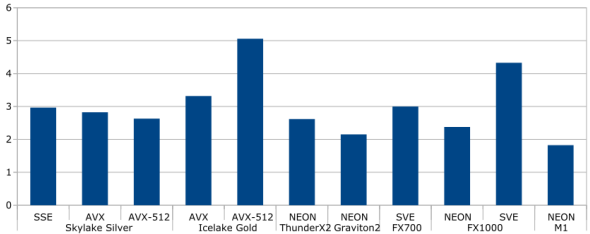
Codes de test

4 versions pour chaque code: scalar, NEON, OpenMP, OpenMP+NEON (excepté la version OpenBLAS):

- Traitement d'image:
  - **grayscale**
  - yuv2rgb
- Normalisation de vecteur
- Multiplication matricielle:
  - **Non optimisé**: pas de vectorisation
  - **Non optimisé NEON**: NEON intrinsics
  - **OpenBLAS**: instructions assembleur NEON, blocking (optimisation de cache), unrolling (optimisation de saut)

unités SIMD et vectorisation

- Codes vectorisés à l'aide d'intrinsics
- Vectorisation automatique non efficace:
  S. Jubertie, F. Dupros, F. De Martin: *Vectorization of a spectral finite-element numerical kernel, WPMVP 2018*
  S. Jubertie, I. Masliah, J. Falcou: Data layout and SIMD abstraction layers: decoupling interfaces from implementations, HPCS 2018



Grayscale scalar

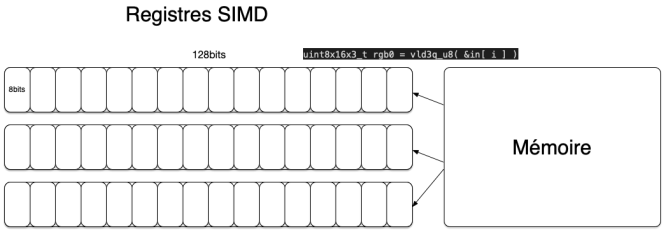
```
#pragma omp parallel for
for( std::size_t i = 0 ; i < out.size() ; ++i )
{
  // out = ( 307 * R + 604 * G + 113 * B ) / 1024
  out[ i ] = ( 307 * in[ 3 * i + 0 ]
             + 604 * in[ 3 * i + 1 ]
             + 113 * in[ 3 * i + 2 ]
             ) >> 10;
}
```

Grayscale NEON, plus de 50 lignes de code

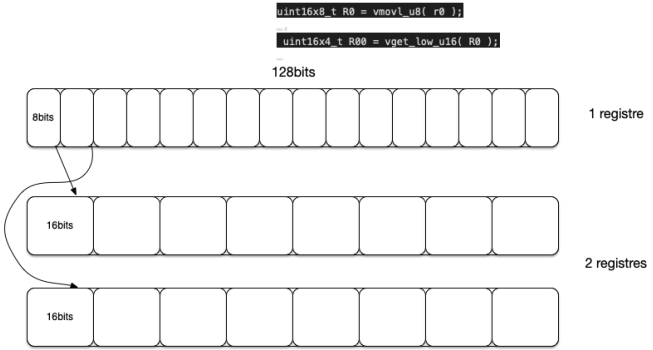
```
#pragma omp parallel for
for( i = 0 ; i < in.size() / 48 * 48 ; i+=48 )
{ uint8x16x3_t rgb0 = vld3q_u8( &in[ i ] );// loads 48 8-bit values, i.e. 16 pixels; deinterleaves them into 3 128 bit NEON registers

  // Separate low/high parts of each register :
  uint8x8_t r0 = vget_low_u8( rgb0.val[ 0 ] );
  uint8x8_t r1 = vget_high_u8( rgb0.val[ 0 ] );
  uint8x8_t g0 = vget_low_u8( rgb0.val[ 1 ] );
  uint8x8_t g1 = vget_high_u8( rgb0.val[ 1 ] );
  uint8x8_t b0 = vget_low_u8( rgb0.val[ 2 ] );
  uint8x8_t b1 = vget_high_u8( rgb0.val[ 2 ] );
  ...
  //computation is done four by four pixels since 8-bit values need to be converted to 32-bit values
  uint32x4_t O00 = vmull_u16( R00, v307 );
  uint32x4_t O01 = vmull_u16( R01, v307 );
  uint32x4_t O10 = vmull_u16( R10, v307 );
  uint32x4_t O11 = vmull_u16( R11, v307 );
  ... }
```

Pour 16 pixels RBG = 48 octets en SIMD on fait 12 mul (48 sinon), 8 add (32 sinon) et 4 shift (16 sinon)

Vectorisation de Grayscale

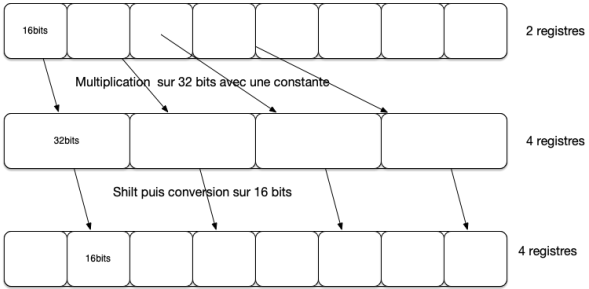


Vectorisation de Grayscale



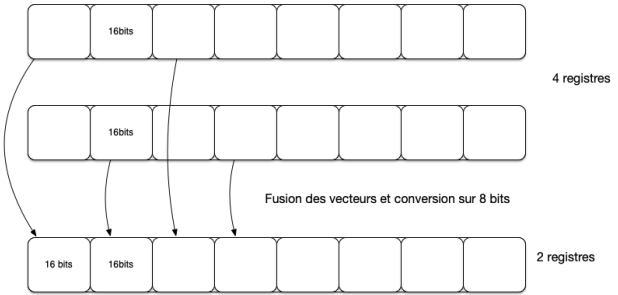
Vectorisation de Grayscale

```
uint16x4_t R00 = vget_low_u16( R0 );
…
uint32x4_t O00 = vmull_u16( R00, v307 );
…
uint16x4_t P00 = vshrn_n_u32( O00, 10 );
```

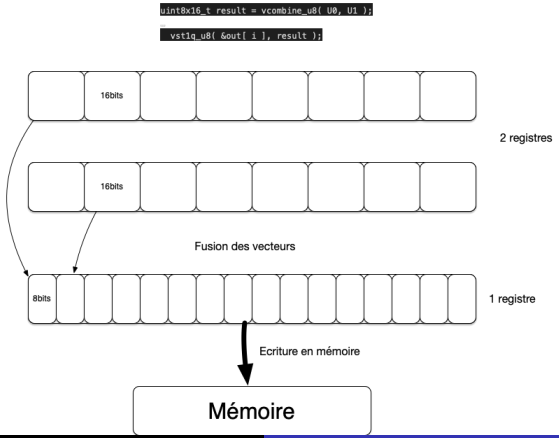


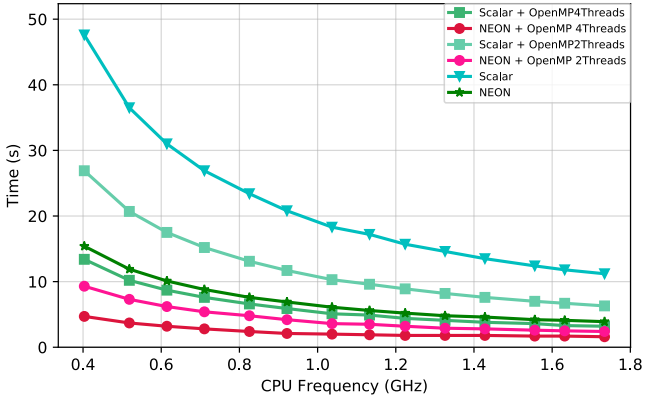
Vectorisation de Grayscale

```
uint16x8_t T0 = vcombine_u16( P00, P01 );
...
uint8x8_t U0 = vmovn_u16( T0 );
...
```

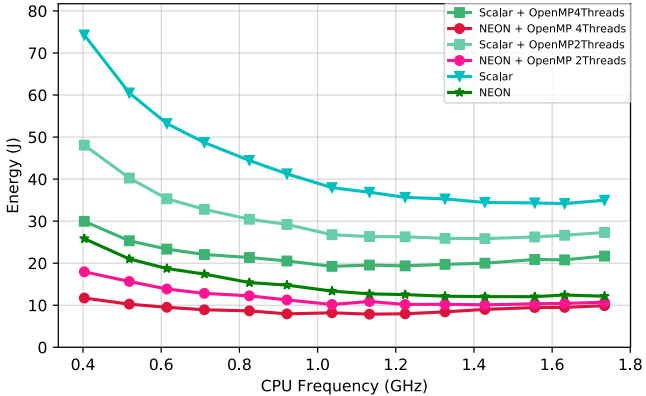


Vectorisation de Grayscale

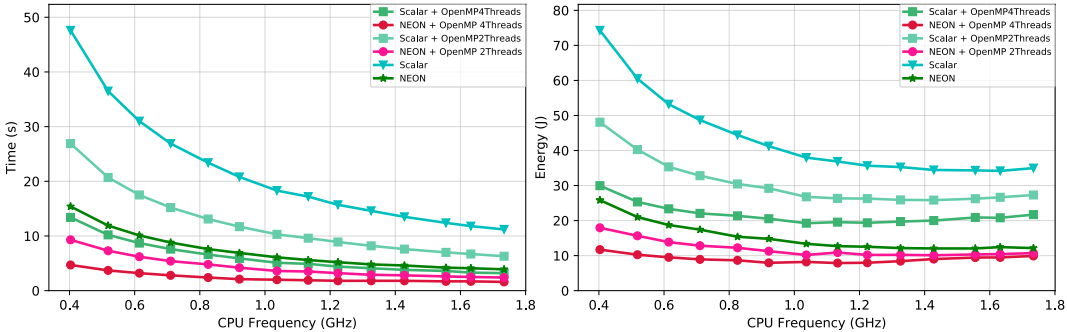




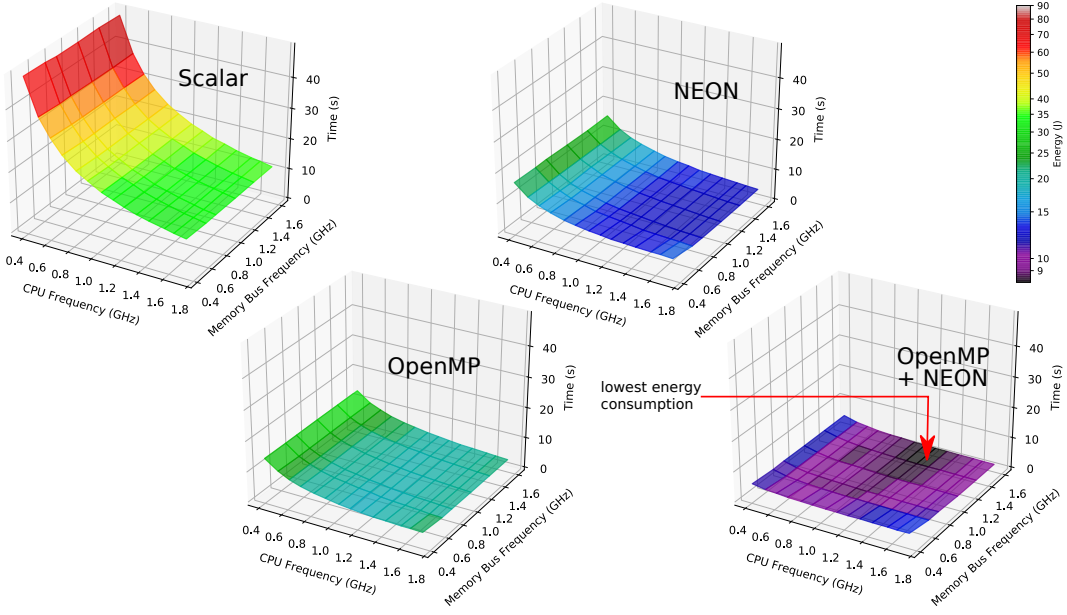
- Fréquence mémoire à 1.6GHz, seules modifications: CPU freq. et nombre de cœurs.
- 1 thread NEON $\approx$ 10-15% plus lent que 4 threads en scalaire
- 4 threads scalaire & 1 thread NEON $\approx$ 2.5-3.5x plus rapide qu'un thread scalaire
- 4 threads NEON $\approx$ 6.0-10.0x plus rapide qu'un thread scalaire
- 4 threads NEON: fréquence x4 $\rightarrow$ seulement une accélération de 2.5
- La bande passante pour une fréquence mémoire donnée n'augmente même pas de 50% en passant à 4 coeurs $\rightarrow$ donc on se rapproche vite d'un scénario memory bound.



- 1 thread NEON $\approx$ 3x moins de consommation d'énergie qu'avec un 1 thread scalaire
- 4 threads scalar $\approx$ 1.5x moins de consommation d'énergie qu'avec un 1 thread scalaire
- la consommation d'énergie la plus basse est avec une fréquence CPU autour de 1.1GHz
- la consommation d'énergie tend à s'élever au-dessus de 1.1GHz



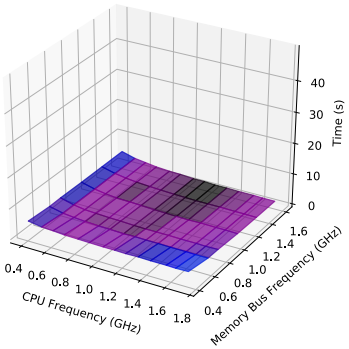
1. 4 threads scalaires et 1 thread NEON $\approx$ performances similaires pour toutes les fréquences...
   ... mais l'écart de consommation d'énergie augmente avec la fréquence.
2. 4 threads NEON $\approx$ 2x plus rapide que 4 threads | NEON
3. Améliorer les temps d'exécution $\rightarrow$ réduit la consommation énergétique...
   ... jusqu'à certaines fréquences.


Scalar
NEON
OpenMP
OpenMP
+ NEON
lowest energy
consumption
Time (s)
Energy (J)
CPU Frequency (GHz)
Memory Bus Frequency (GHz)

Remarques sur grayscale

4 threads NEON → Memory-bound

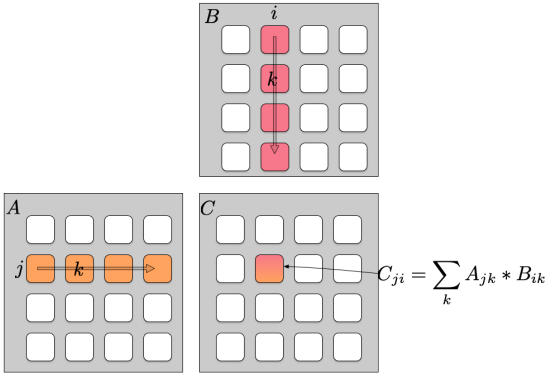
- Aucune amélioration des performances au-dessus de la fréquence CPU 1.2GHz.
- Consommation d'énergie la plus basse à la fréquence Mémoire Maximale.
- Pousser la fréquence processeur à plus de 1,2 GHz → gaspillage d'énergie



Matmul naïf

Code naïf

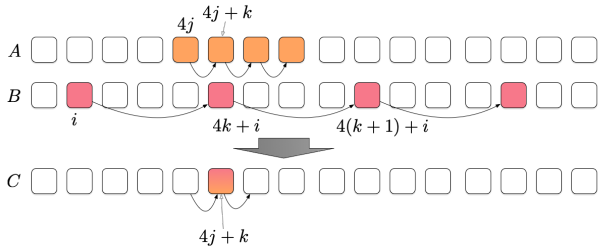
```
#pragma omp parallel for
for( std::size_t j = 0 ; j < dim ; ++j )
{
  for( std::size_t i = 0 ; i < dim ; ++i )
  {
    for( std::size_t k = 0 ; k < dim ; ++k )
    {
m_C[j*dim+i]+=m_A[j*dim+k]*m_B[k*dim+i];
    }
  }
}
```



Matmul naïf

Code naïf

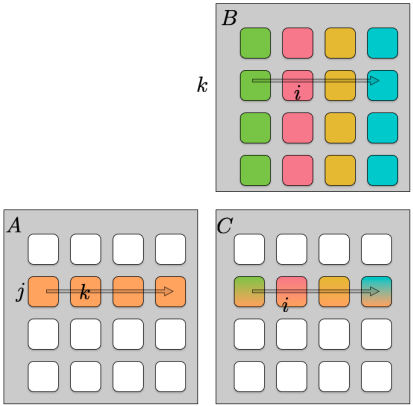
```
#pragma omp parallel for
for( std::size_t j = 0 ; j < dim ; ++j )
{
for( std::size_t i = 0 ; i < dim ; ++i )
{
   for( std::size_t k = 0 ; k < dim ; ++k )
   {
m_C[j*dim+i]+=m_A[j*dim+k]*m_B[k*dim+i];
   }
}
}
```



Matmul non optimisée

Code Cache-Oblivious

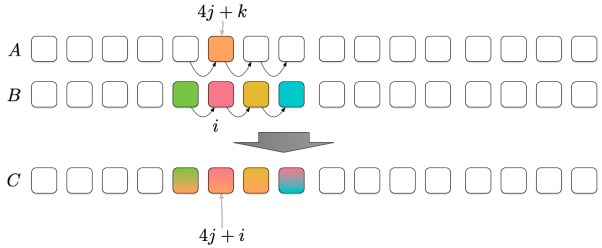
```
#pragma omp parallel for
for( std::size_t j = 0 ; j < dim ; ++j )
{
  for( std::size_t k = 0 ; k < dim ; ++k )
  {
    auto f_a = m_A[ j * dim + k ];
    for( std::size_t i = 0 ; i < dim ; ++i )
    {
      m_C[ j*dim+i ] += f_a * m_B[ k*dim+i ];
    }
  }
}
```



Matmul non optimisée

Code Cache-Oblivious

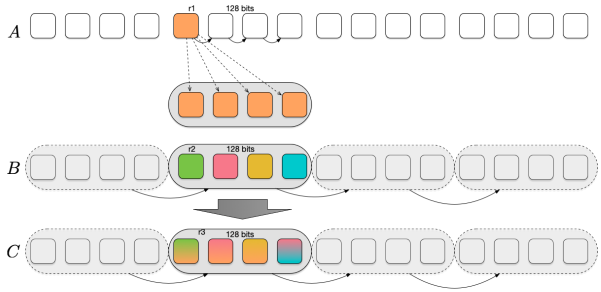
```
#pragma omp parallel for
for( std::size_t j = 0 ; j < dim ; ++j )
{
  for( std::size_t k = 0 ; k < dim ; ++k )
  {
    auto f_a = m_A[ j * dim + k ];
    for( std::size_t i = 0 ; i < dim ; ++i )
    {
      m_C[ j*dim+i ] += f_a * m_B[ k*dim+i ];
    }
  }
}
```

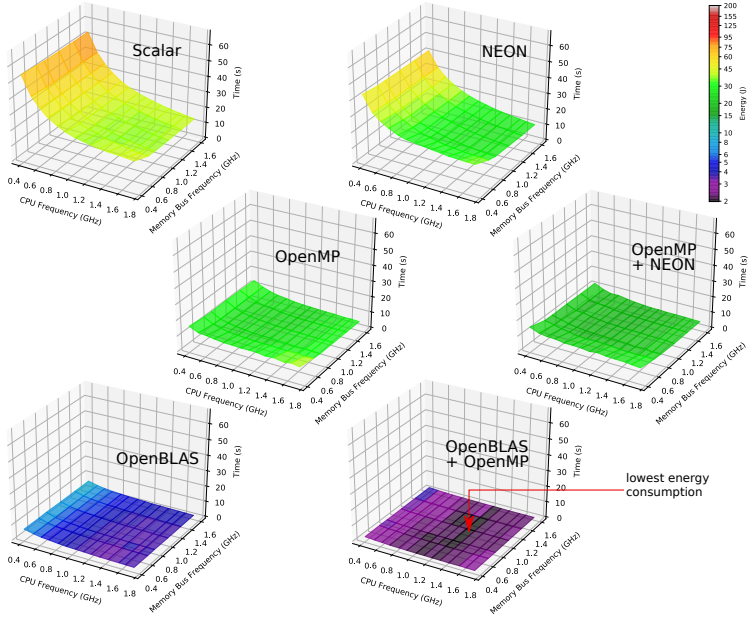


Matmul NEON

Code Cache-Oblivious

```
#pragma omp parallel for
for( std::size_t j = 0 ; j < dim ; ++j )
{
  auto js = j * dim;

  for( std::size_t k = 0 ; k < dim ; ++k )
  {
    auto ks = k * dim;
    auto r1 = vld1q_dup_f32( &m_A[ js + k ] );

    for( std::size_t i = 0 ; i < dim ; i+=4 )
    {
      auto r3 = vld1q_f32( &m_C[ js + i ] );
      auto r2 = vld1q_f32( &m_B[ ks + i ] );
      r3 = vfmaq_f32( r3, r1, r2 );
      vst1q_f32( &m_C[ js + i ], r3 );
    }
  }
}
```




Scalar
NEON
OpenMP
OpenMP
+ NEON
OpenBLAS
OpenBLAS
+ OpenMP
lowest energy
consumption
Time (s)
Energy (J)
CPU Frequency (GHz)
Memory Bus Frequency (GHz)

Conclusion

1. Les meilleures performances: fréquences CPU et mémoire maximales. (évident)...
   ...Mais les meilleures performances coûtent beaucoup d'énergie (en particulier les codes memory-bound).
   $\rightarrow$ Meilleur rapport perf / énergie autour de 1,2 GHz (CPU),
   $\rightarrow$ la fréquence mémoire dépend du scénario de mémoire / de calcul
2. Codes optimisés (cache): accélération similaire NEON vs 4 threads scalaire (FP32) mais moins d'énergie.
   L'écart augmente avec la fréquence du CPU.
3. Une vectorisation efficace est plus intéressante pour la consommation d'énergie que le multithreading.

Vectorisez toujours votre code (vérifiez au moins si le compilateur vectorise) !!!
... mais nécessite presque toujours un effort de développement

Perspective

- Tests sur d'autres architectures (Big.Little, x86)
- Tests avec des GPU intégrés ou sur carte graphique
-
  - PCAT Power Capture Analysis Tool (Outil d'analyse Energétique de Nvidia)
  - Plateforme expérimentale LIFO
- Tester des codes plus complexes: Instrumenter un noyau FEM (extrait d'Efispec BRGM)
- Intégrer le contrôle dans l'application