Parallélisation automatique de boucles pour processeurs vectoriels

Félix-Antoine Ouellet

Université de Sherbrooke

18 Septembre 2014

Plan

Loi de Moore

"Le nombre de transistors dans les microprocesseurs double tous les 18 mois."

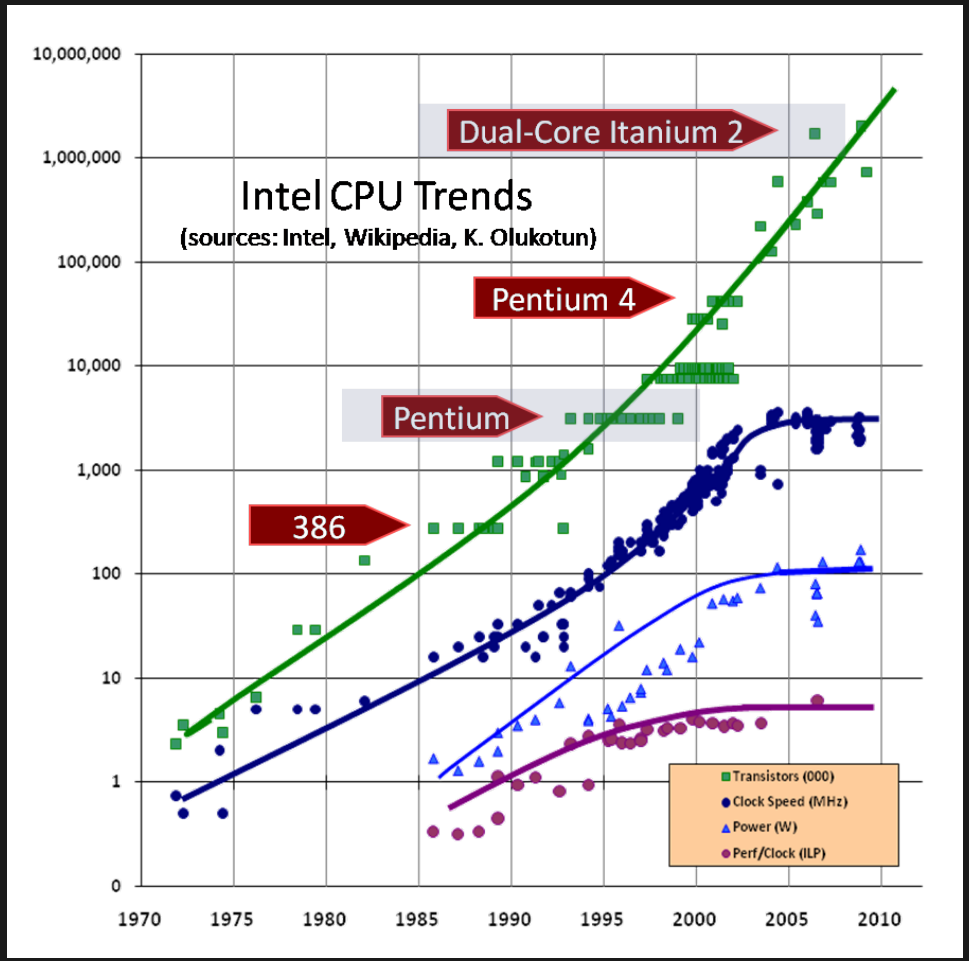
- Loi de Moore

Constat de l'industrie

"The free lunch is over"

- Herb Sutter

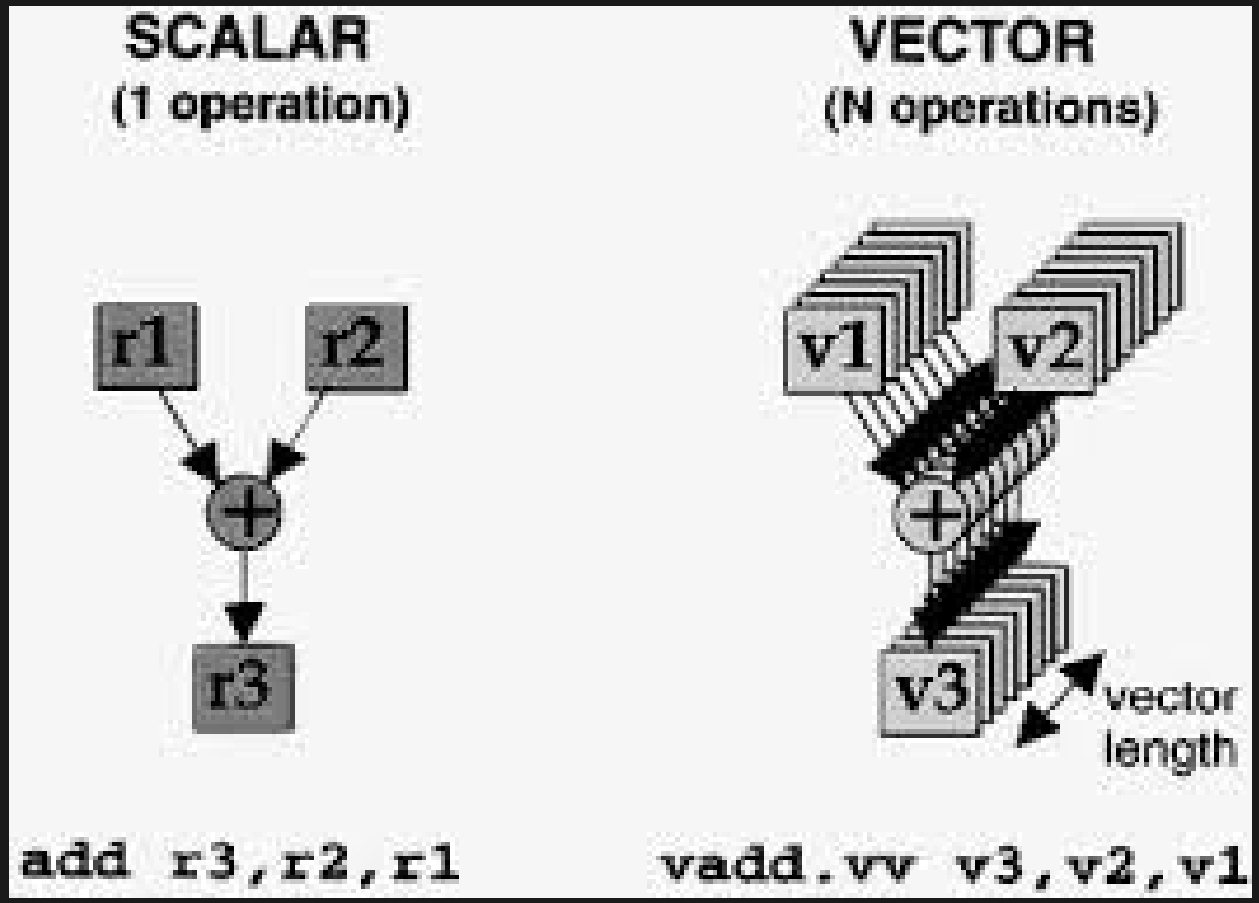
Constat de l'industrie



Avenues possible

- Processeurs multi-coeurs
- Accélérateurs
- Processeurs vectoriels

Processeurs vectoriels



Plan

Code Séquentiel

Somme des éléments de vecteurs

```
int reduction = 0;
for (int i = 0; i < 100; ++i) {
  reduction += A[i];
}
```

Code Vectoriel

Somme des éléments de vecteurs

```
int reductionTab[] = { 0, 0 };
for (int i = 0; i < 100; i+=8) {
  reductionTab[0] += A[i:i+3];
  reductionTab[1] += A[i+4:i+7];
}
int reduction = 0;
for (int i = 0; i < 2; ++i) {
  reduction += reductionTab[i];
}
for (int i = 96; i < 100; ++i) {
  reduction += A[i];
}
```

Plan

Notions de base

Classification des dépendences mémoire

```
void fct() {
  int a = 20;
  int b = a;
  /* ... */
  int c = d;
  int d = e;
}
```

Notions de base

Classification des dépendences mémoire

```
void fct() {
  int a = 20;
  int b = a;
  /* ... */
  int c = d;
  int d = e;
}
```

Vraie dépendence
(Lecture après écriture)

Notions de base

Classification des dépendences mémoire

```
void fct() {
  int a = 20;
  int b = a;
  /* ... */
  int c = d;
  int d = e;
}
```

Vraie dépendence
(Lecture après écriture)

Anti-dépendence
(Écriture après lecture)

Notions de base

Classification des dépendences de boucles

```
for (int i = 0; i < 100; ++i) {
  A[i] = B[i] + C[i];
  D[i] = A[i] + 10;
}

for (int i = 0; i < 100; ++i) {
  A[i+1] = A[i] + B[i];
}
```

Notions de base

Classification des dépendences de boucles

```
for (int i = 0; i < 100; ++i) {
  A[i] = B[i] + C[i];
  D[i] = A[i] + 10;
}
```

Dépendence indépendante de la boucle

```
for (int i = 0; i < 100; ++i) {
  A[i+1] = A[i] + B[i];
}
```

Notions de base

Classification des dépendences de boucles

```
for (int i = 0; i < 100; ++i) {
  A[i] = B[i] + C[i];
  D[i] = A[i] + 10;
}
```

Dépendence indépendante de la boucle

```
for (int i = 0; i < 100; ++i) {
  A[i+1] = A[i] + B[i];
}
```

Dépendence portée par la boucle

Légalité

Parallélisation des instructions

```
int reduction = 0;
for (int i = 0; i < 100;
     ++i) {
  reduction += A[i];
}
```

Légalité
Parallélisation des instructions

✓ Pas d'appels de fonctions

```
int reduction = 0;
for (int i = 0; i < 100;
     ++i) {
  reduction += A[i];
}
```

Légalité

Parallélisation des instructions

```
int reduction = 0;
for (int i = 0; i < 100;
     ++i) {
  reduction += A[i];
}
```

- ✓ Pas d'appels de fonctions
- ✓ Opération parallélisable

Légalité

Parallélisation des instructions

```
int reduction = 0;
for (int i = 0; i < 100;
     ++i) {
  reduction += A[i];
}
```

- ✓ Pas d'appels de fonctions
- ✓ Opération parallélisable
- ✓ Types des paramètres parallélisables

Légalité

Parallélisation des instructions

```
int reduction = 0;
for (int i = 0; i < 100;
     ++i) {
  reduction += A[i];
}
```

- ✓ Pas d'appels de fonctions
- ✓ Opération parallélisable
- ✓ Types des paramètres parallélisables
- ✓ Type de retour parallélisable

Légalité
Parallélisation de la mémoire

```
int reduction = 0;
for (int i = 0; i < 100; ++i) {
  reduction += A[i];
}
```

Légalité
Parallélisation de la mémoire

```
int reduction = 0;
for (int i = 0; i < 100; ++i) {
  reduction += A[i];
}
```

✓ Pas de chevauchement d'accès mémoire

Légalité
Parallélisation de la mémoire

```
int reduction = 0;
for (int i = 0; i < 100; ++i) {
  reduction += A[i];
}
```

- ✓ Pas de chevauchement d'accès mémoire
- × Présence de dépendences mémoire

Profitabilité

- Lié à l'architecture physique
- Coût version séquentielle VS Coût version vectorielle

Séquentielle	Vectorielle
Coût add i64	Coût add <2 x i64>
Coût load i64	Coût load <2 x i64>

- Meilleur facteur de déroulement

Reconnaissance d'idiomes

Théorie

- But: Agir en présence d'une situation connue
- Exemples:
  - Induction
  - Réduction

Reconnaissance d'idiomes

Pratique

```
int reduction = 0;
int reductionTab[2];
for (int i = 0; i < 2; ++i) {
  reductionTab[i] = 0;
  for (int j = i; j < 100; j+=2) {
    reductionTab[i] += A[j];
  }
  reduction += reductionTab[i];
}
```

Distribution de boucle

Théorie

- But: Regrouper les calculs similaires
- Moyen: Produire plusieurs boucles à partir de la boucle originale

Distribution de boucle

Pratique

```
int reductionTab[] = { 0, 0 };
for (int i = 0; i < 2; ++i) {
  for (int j = i; j < 100; j+=2) {
    reductionTab[i] += A[j];
  }
}
int reduction = 0;
for (int i = 0; i < 2; ++i) {
  reduction += reductionTab[i];
}
```

Inter-échange de boucles

Théorie

- But: Optimiser la localité de référence
- Moyen : Échanger les variables d'induction des boucles ciblées

Inter-échange de boucles

Pratique

```
int reductionTab[] = { 0, 0 };
for (int j = 0; j < 100; j+=2) {
  for (int i = j; i < min(j+2, 100); ++i) {
    reductionTab[i-j] += A[j];
  }
}
int reduction = 0;
for (int i = 0; i < 2; ++i) {
  reduction += reductionTab[i];
}
```

Vectorization
Théorie

- But: Exploiter les registres et opérations vectoriels disponibles
- Moyen : Générer du code vectoriel

Vectorization

Pratique

```
int reductionTab[] = { 0, 0 };
for (int j = 0; j < 100; j+=2) {
  for (int i = j; i < min(j+2, 100); ++i) {
    reductionTab[i-j] += A[j:j+3];
  }
}
int reduction = 0;
for (int i = 0; i < 2; ++i) {
  reduction += reductionTab[i];
}
```

Déroulage de boucle
Théorie

- But: Réduire le temps d'exécution d'une boucle
- Moyen : Expliciter les calculs dans une boucle
- Attention, on choisit de prendre plus de mémoire pour gagner en vitesse d'exécution

Déroulage de boucle

Pratique

```
int reductionTab[] = { 0, 0 };
for (int i = 0; i < 100; i+=8) {
  reductionTab[0] += A[i:i+3];
  reductionTab[1] += A[i+4:i+7];
}
int reduction = 0;
for (int i = 0; i < 2; ++i) {
  reduction += reductionTab[i];
}
for (int i = 96; i < 100; ++i) {
  reduction += A[i];
}
```

Plan

Pointeurs

```
void bar(float *A, float* B, float K, int n) {
  for (int i = 0; i < n; ++i)
    A[i] *= B[i] + K;
}
```

Superword Level Parallelism

```
void foo(int a1, int a2, int b1, int b2, int *A)
{
  A[0] = a1*(a1 + b1)/b1 + 50*b1/a1;
  A[1] = a2*(a2 + b2)/b2 + 50*b2/a2;
}
```

Plan

Conclusion

- L'autovectorization c'est bien, mais c'est limité
- L'architecture change donc la programmation doit changer