# Roadmap

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
**Optimizations**
Memory & caches

**C:**

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

**Java:**

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

**Assembly language:**

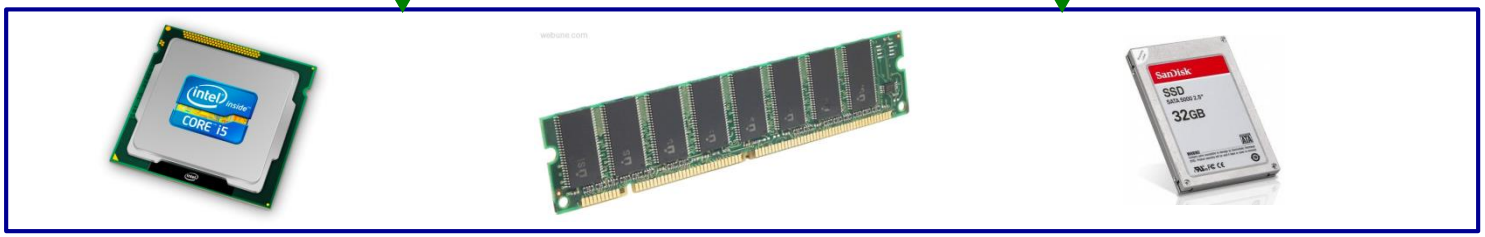
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

**Machine code:**

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100001111l
```

**OS:**


**Computer system:**



# Cours 7: Optimisations de programme

- **Quelques optimisations utiles :**
  - Code motion/pré-calcul
  - Réduction en force
  - Partage de sous-expressions communes
  - Élimination des appels de procédure inutiles
- **Blocage :**
  - Appels de procédure
  - Aliasing
- **Parallélisme au niveau d'instruction**

# Performance

- **Il faut optimiser votre code au différents niveaux :**
  - algorithmes, structures de données, procédures, et boucles
- **Il faut comprendre le système pour optimiser la performance**
  - Comment les programmes sont compilés et exécutés ?
  - Comment mesurer la performance du programme et identifier les goulots ?
  - Comment améliorer la performance sans détruire la compréhension, la modularité et la généralité du code ?

# Optimisations du compilateur

- **Fournir un mapping efficace entre programme et machine**
  - Allocation des registres
  - Code scheduling
  - Élimination du code mort
  - Élimination des inefficacités mineurs
- **Mais n'améliore pas votre code de manière spectaculaire**
  - Au programmeur de choisir des meilleurs algorithmes
  - big-O savings vs constant factors
- **Avoir des difficultés pour résoudre les blocages**
  - Alias potentiel
  - Effets de bornes potentiels des procédures
  - Quand il y a des doutes, le compilateur doit être conservatif !

# Même comportement ?

```
void twiddle1(long *xp, long *yp) {
    *xp += *yp;
    *xp += *yp;
}
```

```
void twiddle2(long *xp, long *yp) {
    *xp += 2* *yp;
}
```

# Appel de fonction : effet de borne

```
long f();

long fun1() {
    return f() + f() + f() + f();
}

long fun2() {
    return 4*f();
}
```

# Cours 7: Optimisations de programme

- **Quelques optimisations utiles :**
  - Code motion/pré-calcul
  - Réduction en force
  - Partage de sous-expressions communes
  - Élimination des appels inutiles de procédure
- **Blocage :**
  - Appels de procédure
  - Aliasing
- **Parallélisme au niveau d'instruction**

# Code Motion

- **Réduire la fréquence d'exécution d'un calcul :**
  - S'il produit toujours le même résultat
  - Spécialement en dehors d'une boucle

```
void set_row(double *a, double *b,
long i, long n){
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

# Code Motion

- **Réduire la fréquence d'exécution d'un calcul :**
  - S'il produit toujours le même résultat
  - Spécialement en dehors d'une boucle

```
void set_row(double *a, double *b,
long i, long n){
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```



```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

# Réduction en force

- Remplacer des opérations couteuses par celles plus simples
- Décalage, addition au lieu de multiplication ou division (Intel Nehalem, multiplication des entiers demandent 3 cycles CPU)

```
16*x  -->   x << 4
```

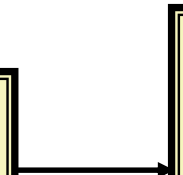
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

# Réduction en force

- Remplacer des opérations couteuses par celles plus simples
- Décalage, addition au lieu de multiplication ou division (Intel Nehalem, multiplication des entiers demandent 3 cycles CPU)

```
16*x  -->   x << 4
```

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```



```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

# Partage des sous-expressions communes

- Réutiliser les sous-expressions
- Les compilateurs ne sont pas si sophistiqués …

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
leaq    1(%rsi), %rax  # i+1
leaq    -1(%rsi), %r8  # i-1
imulq   %rcx, %rsi     # i*n
imulq   %rcx, %rax     # (i+1)*n
imulq   %rcx, %r8      # (i-1)*n
addq    %rdx, %rsi     # i*n+j
addq    %rdx, %rax     # (i+1)*n+j
addq    %rdx, %r8      # (i-1)*n+j
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
imulq    %rcx, %rsi  # i*n
addq     %rdx, %rsi  # i*n+j
movq     %rsi, %rax  # i*n+j
subq     %rcx, %rax  # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

# Cours 7: Optimisations de programme

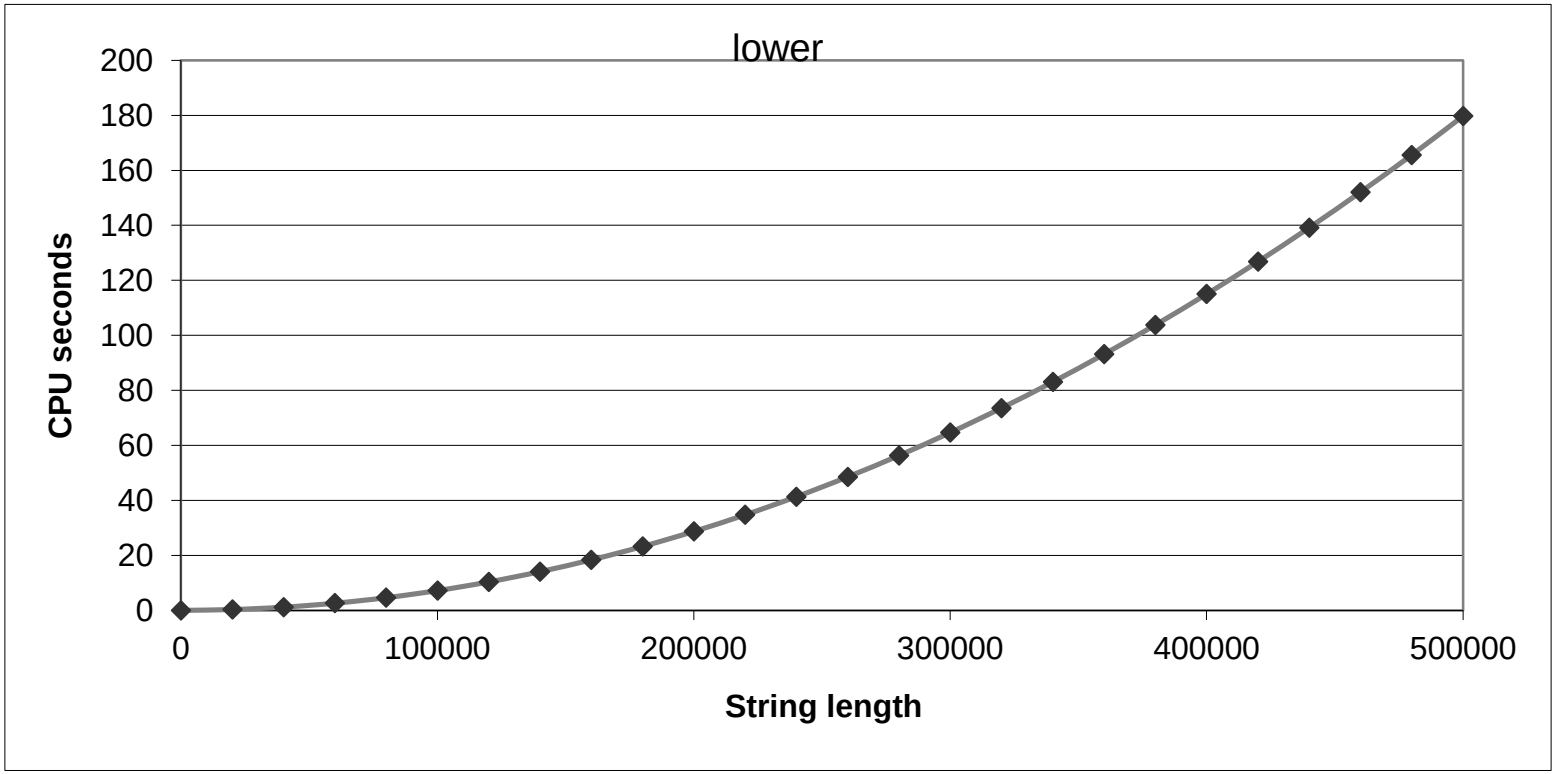
- **Quelques optimisations utiles :**
  - Code motion/pré-calcul
  - Réduction en force
  - Partage de sous-expressions communes
  - Élimination des appels inutiles de procédure
- **Blocage :**
  - Appels de procédure
  - Aliasing
- **Parallélisme au niveau d'instruction**

# Blocage d'optimisation: Appel de procédure

- **Procédure pour convertir une chaîne de caractères en minuscules :**

```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

# Performance quadratique

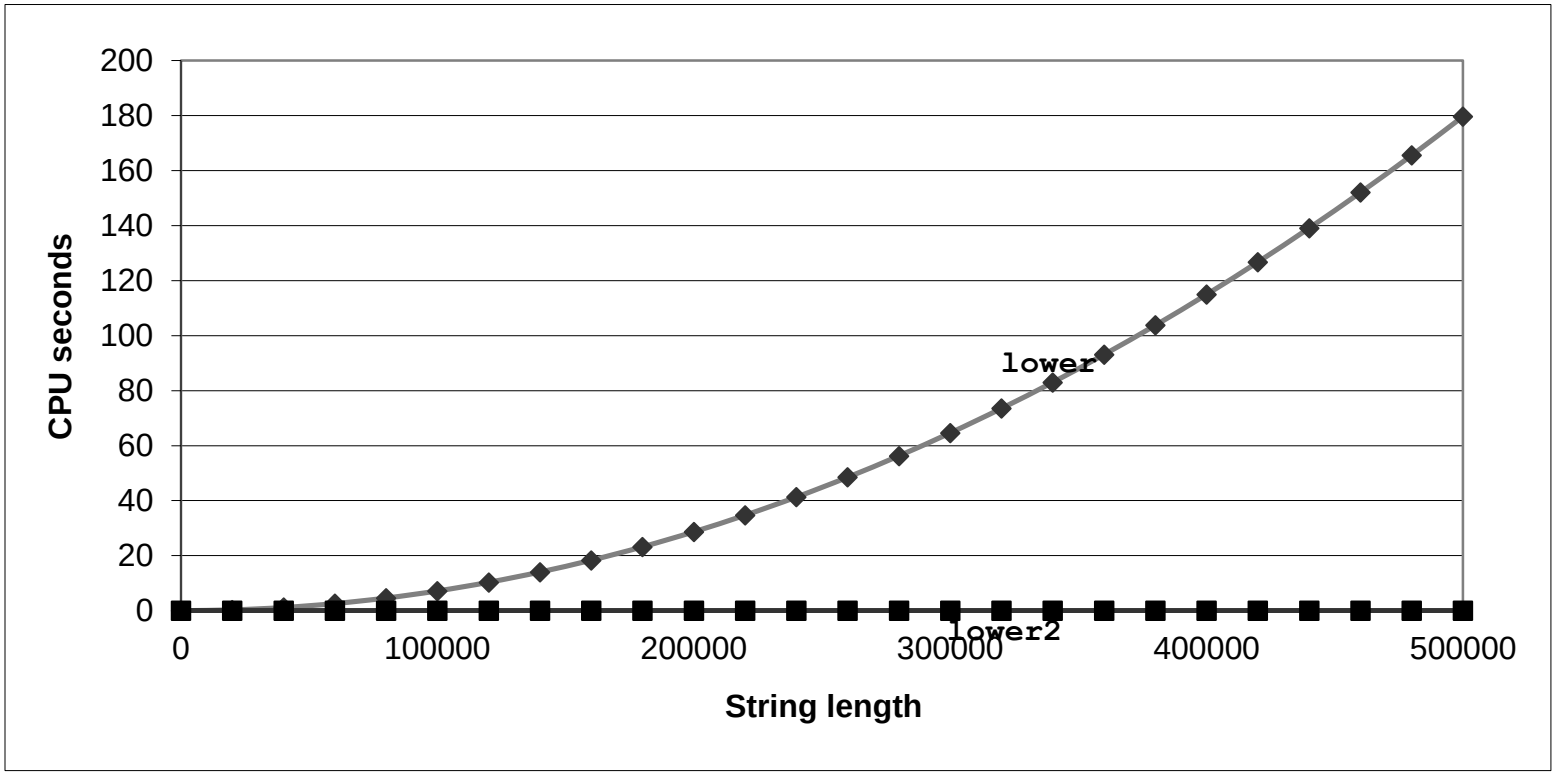


# Améliorer la performance

```
void lower2(char *s){
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

# Performance

- Quadratique vs linéaire !




**1.034.000 éléments : 17 minutes vs 2 millisecondes**

# Blocage d'optimisation: Appel de procédure

- ***Pourquoi le compilateur ne peut pas enlever `strlen` en dehors de la boucle ?***
  - Procédure peut avoir des effets de bornes
  - Fonction peut retourner des valeurs différentes
- **Warning:**
  - Compilateur traite des appels de procédure comme des boites noires
  - Optimisations faibles
- **Solution :**
  - Utiliser `inline`
    - gcc –O1
  - Faire vous-même code motion !

```
int lencnt = 0;
size_t strlen(const char *s){
    size_t length = 0;
    while (*s != '\0') {
       s++; length++;
    }
    lencnt += length;
    return length;
}
```

# Aliasing ?

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
# sum_rows1 inner loop
.L53:
        addsd   (%rcx), %xmm0           # FP add
        addq    $8, %rcx
        decq    %rax
        movsd   %xmm0, (%rsi,%r8,8)     # FP store
        jne     .L53
```

- La boucle intérieure doit mettre à jour `b[i]` à chaque itération
- Pourquoi le compilateur ne peut pas optimiser ?

# Aliasing ?

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

**Value of B:**

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

- La boucle d'intérieur doit mettre à jour `b[i]` à chaque itération
- Doit considérer que ces mises à jour peuvent affecter le comportement du programme

# Enlever l'alias

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
# sum_rows2 inner loop
.L66:
        addsd   (%rcx), %xmm0   # FP Add
        addq    $8, %rcx
        decq    %rax
        jne     .L66
```

- Pas besoin de stocker les résultats intermédiaires

# Blocage d'optimisation: Aliasing

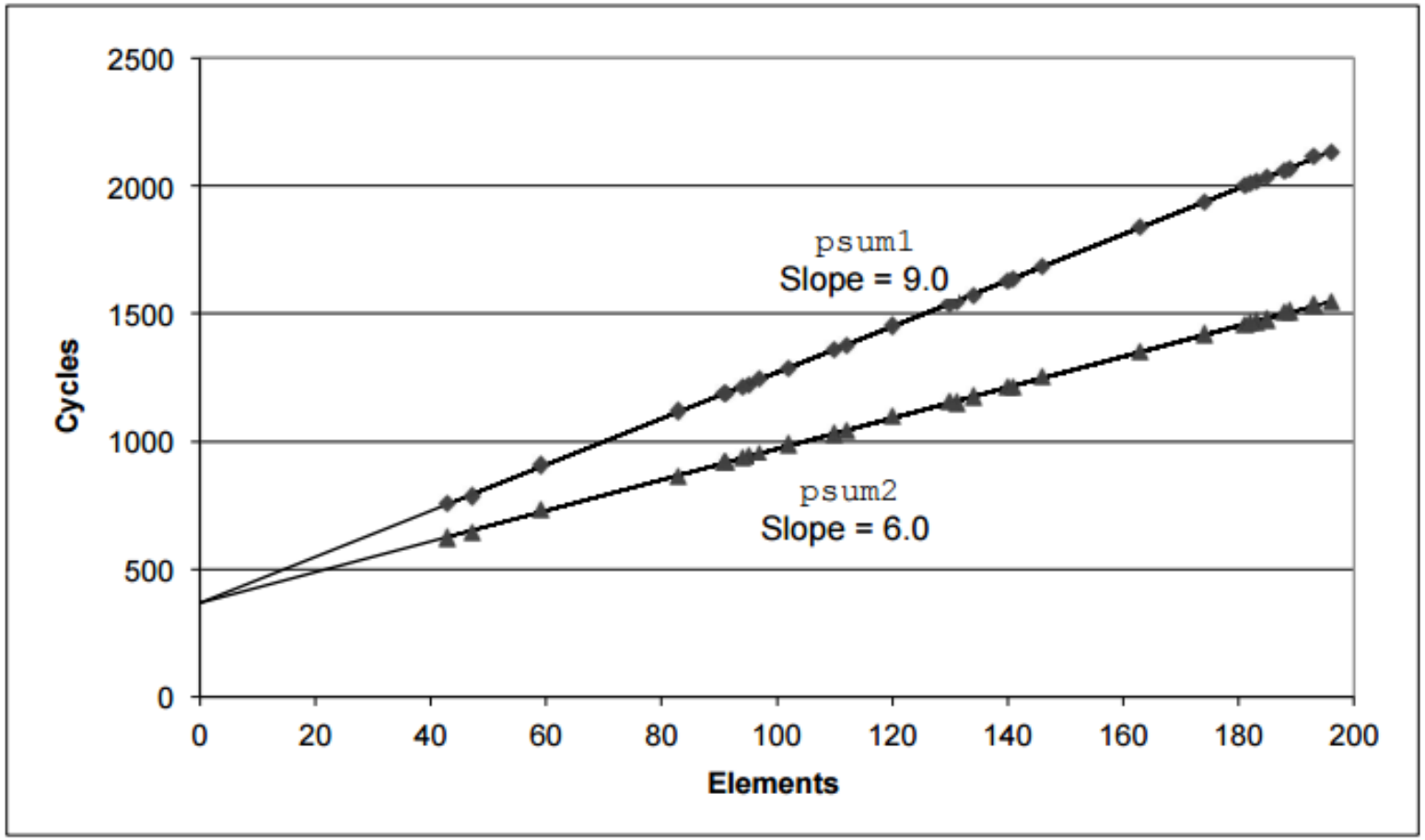
- **Aliasing**
  - Deux références mémoires pointent sur la même localisation
  - Facile à avoir en C à cause de :
    - Arithmétique sur les adresses
    - Accès direct aux structures de stockage
  - Ajouter des variables locales
    - Accumuler dans une boucle
    - Dire au compilateur de ne pas tester pour aliasing

# Loop unrolling : exemple

```
void psum1(float a[], float p[], long n) {
    long i;
    p[0] = a[0];
    for (i = 1; i < n; i++)
        p[i] = p[i-1] + a [i];
}
```

```
void psum2(float a[], float p[], long n) {
    long i;
    p[0] = a[0];
    for (i = 1; i < n-1; i+=2) {
        float mid_val = p[i-1] + a [i];
        p[i] = mid_val;
        p[i+1] = mid_val + a[i+1];
    }
    if (i < n)  // finish remaining element
        p[i] = p[i-1] + a[i];
}
```

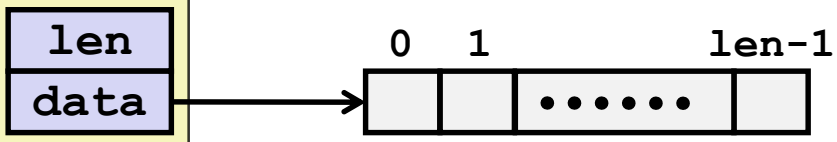
# Mesure de performance :
# CPE = cycles per élément




**T = CPE*n + Overhead**
**psum1 : 9.0*n + 368**
**psum2 : 6.0*n + 368**

# Cas d'étude

```
/* abstract data type for vectors */
typedef struct{
    long len;
    data_t *data;
} * vec_ptr;
```




```
/* retrieve vector element and store at val */
int get_vec_element(vec_ptr v, long index, data_t *val){
    if (index < 0 || index >= v->len)
        return 0;
    *val = v->data[index];
    return 1;
}
```

# Benchmark

```
void combine1(vec_ptr v, data_t *dest){
    long i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

**Calculer la somme ou le produit des éléments du vecteur**

- **Types de données (`data_t`)**
  - `int`
  - `float`
  - `double`
- **Opération**
  - `OP : + ou *`
- **Élément d'identité**
  - `IDENT : 0 ou 1`

# Performance

```
void combine1(vec_ptr v, data_t *dest){
    long i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

**Calculer la somme ou le produit des éléments du vecteur**

| Method | Integer | | Double | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| combine1 –O1 | 10.12 | 10.12 | 10.17 | 11.14 |

**Intel Core i7 Haswell**

# 1) Éliminer les inefficacités de boucle

- Code motion : déplacer vec_length en dehors de boucle

```
void combine2(vec_ptr v, data_t *dest){
  long i;
  long length = vec_length(v);
  *dest = IDENT;
  for (i = 0; i < length; i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
  }
}
```

# 1) Éliminer les inefficacités de boucle

- Code motion : déplacer vec_length en dehors de boucle

```
void combine2(vec_ptr v, data_t *dest){
  long i;
  long length = vec_length(v);
  *dest = IDENT;
  for (i = 0; i < length; i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
  }
}
```

| Method | Integer | | Double | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| combine1 –O1 | 10.12 | 10.12 | 10.17 | 11.14 |
| combine2 | 7.02 | 9.03 | 9.02 | 11.03 |

# 2) Réduire les appels de procédure

- get_vec_element est déplacé => éviter les bound-checks dans la boucle

```
void combine3(vec_ptr v, data_t *dest){
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  *dest = IDENT;
  for (i = 0; i < length; i++) {
       *dest = *dest OP data[i];
  }
}
```

# 2) Réduire les appels de procédure

- get_vec_element est déplacé => éviter les bound-checks dans la boucle

```
void combine3(vec_ptr v, data_t *dest){
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  *dest = IDENT;
  for (i = 0; i < length; i++) {
       *dest = *dest OP data[i];
  }
}
```

| Method | Integer | | Double | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| combine2 | 7.02 | 9.03 | 9.02 | 11.03 |
| combine3 | 7.17 | 9.02 | 9.02 | 11.03 |

# 3) Eliminer les références mémoires inutiles

- Référence mémoire corresponde à dest est lue et écrite à chaque itération => la remplacer par une variable locale accumulateur (registre)

```
void combine4(vec_ptr v, data_t *dest){
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  for (i = 0; i < length; i++) {
       acc = acc OP data[i];
  }
  *dest = acc;
}
```

# 3) Eliminer les références mémoires inutiles

- Référence mémoire corresponde à dest est lue et écrite à chaque itération => la remplacer par une variable locale accumulateur (registre)

```
void combine4(vec_ptr v, data_t *dest){
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  for (i = 0; i < length; i++) {
       acc = acc OP data[i];
  }
  *dest = acc;
}
```

| Method | Integer | | Double | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| combine3 | 7.17 | 9.02 | 9.02 | 11.03 |
| combine4 | 1.27 | 3.01 | 3.01 | 5.01 |

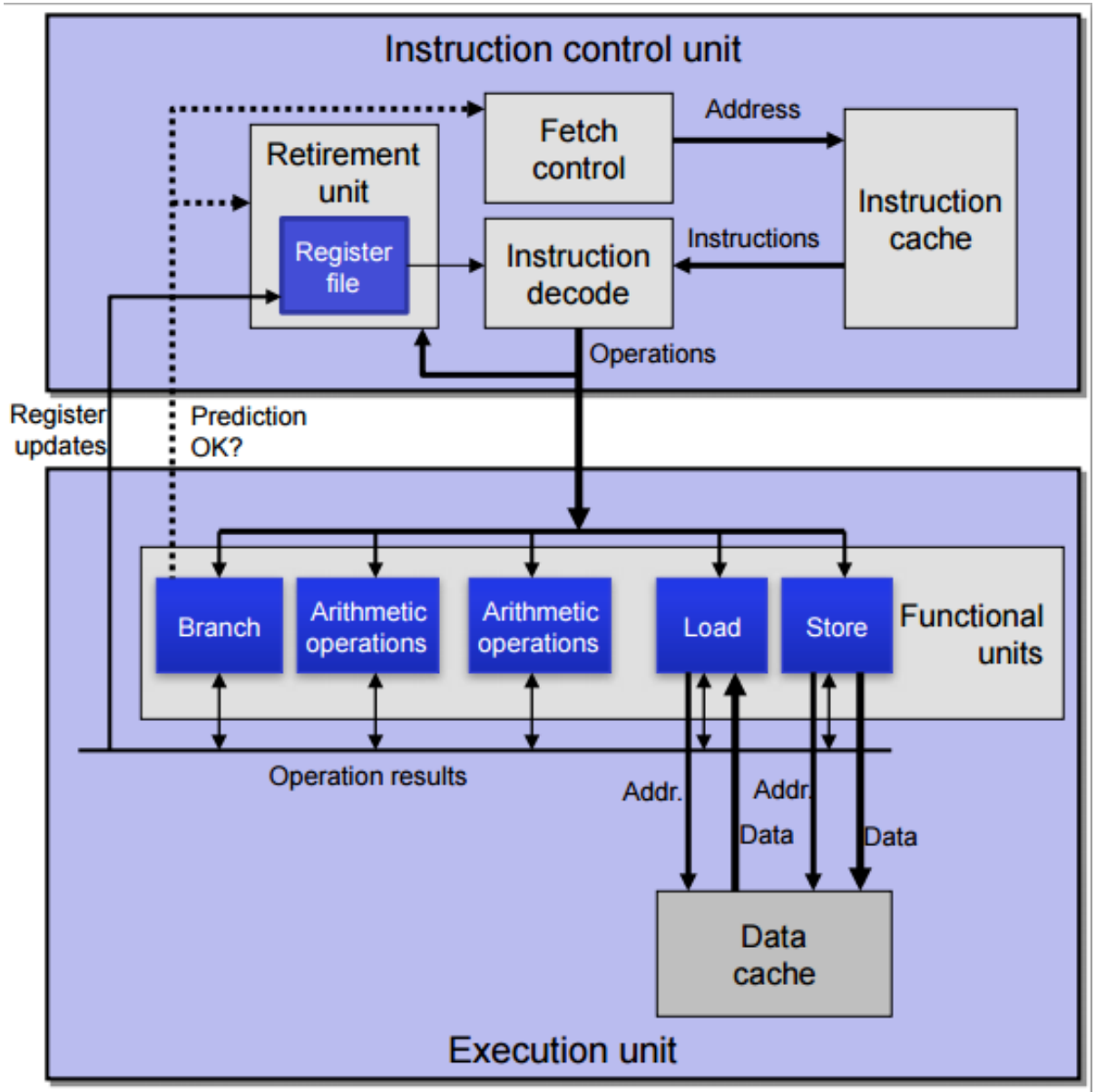
# Cours 7: Optimisations de programme

- **Quelques optimisations utiles :**
  - Code motion/pré-calcul
  - Réduction en force
  - Partage de sous-expressions communes
  - Élimination des appels inutiles de procédure
- **Blocage :**
  - Appels de procédure
  - Aliasing
- **Parallélisme au niveau d'instruction**

# Exploiter parallélisme au niveau d'instruction

- **Les processeurs peuvent exécuter des instructions multiples en parallèle (processeurs superscalars + out of order)**
- **Mais la performance est souvent limitée à cause de dépendance de données**
- **Des transformations simples peuvent avoir de très bonnes améliorations**
  - Mais les compilateurs ne peuvent pas souvent faire ces transformations
  - Pas d'associativité ni de distributivité avec les nombres flottants

# Conception de CPU moderne

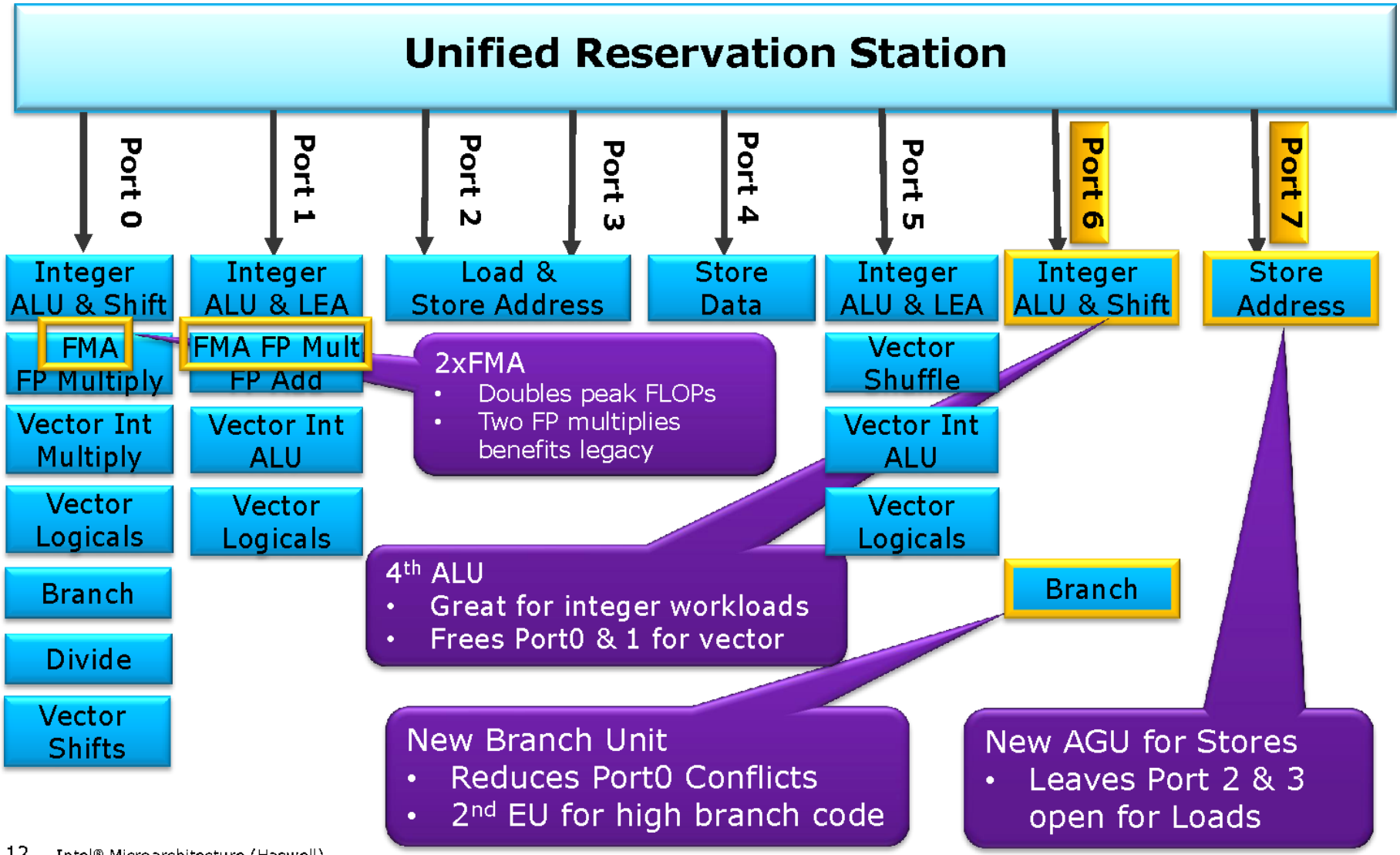


# Intel Core i7 Haswell

| | Integer | | | Floating point | | |
|---|---|---|---|---|---|---|
| Opération | Latence | Issue | Capacité | Latence | Issue | Capacité |
| Addition | 1 | 1 | 4 | 3 | 1 | 1 |
| Multiplication | 3 | 1 | 1 | 5 | 1 | 2 |
| Division | 3-30 | 3-30 | 1 | 3-15 | 3-15 | 1 |

- **Latence (latency) : nombre total de cycles nécessaires pour réaliser l'opération**
- **Issue (issue) : nombre minimal de cycles entre deux opérations indépendantes de même type (pipeline)**
- **Capacité (capacity) : nombre d'opérations pouvant être issues simultanément (nombre d'unités fonctionnelles)**

# Haswell Execution Unit Overview



# Intel Core i7 Haswell

| | Integer | | Floating point | |
|---|---|---|---|---|
| | **Add** | **Mult** | **Add** | **Mult** |
| **combine4** | **1.27** | **3.01** | **3.01** | **5.01** |
| Latency Bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput Bound | 0.50* | 1.00 | 1.00 | 0.50 |

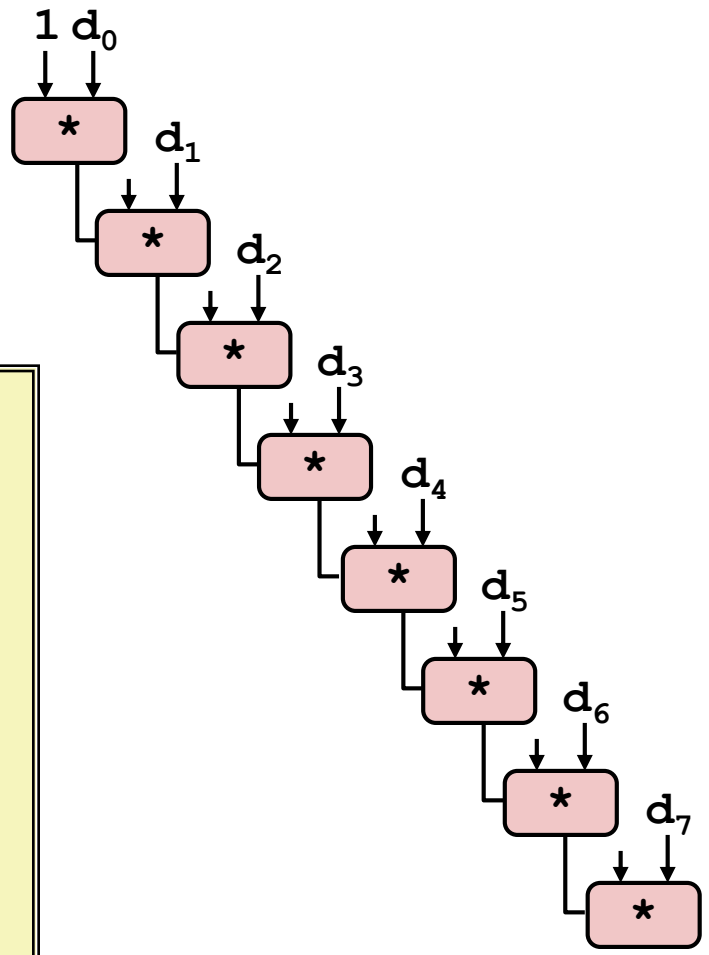
- **Débit (throughput) = Issue /Capacité : nombre d'opérations per cycle**

**4 unités fonc.pour int +**
**2 unités fonc.pour load**

**2 unités fonc.pour FP ***
**2 unités fonc.pour load**

# Flot de données

```
void combine4(vec_ptr v, data_t *dest){
  long i;
  long length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  for (i = 0; i < length; i++) {
       acc = acc OP data[i];
  }
  *dest = acc;
}
```



# Loop Unrolling

```
/* 2 x 1 loop unrolling */
void combine5(vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       acc = (acc OP data[i]) OP data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
       acc = acc OP data[i];
    }
    *dest = acc;
}
```

# Effet de Loop Unrolling

| Method | Integer | | Double | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| combine5 (2 x 1 unrolling) | 1.01 | 3.01 | 3.01 | 5.01 |
| Latency Bound | 1.00 | 3.00 | 3.00 | 5.00 |

- Améliorer l'addition Integer en réduisant la surcharge de boucle (moins d'index + jump)
- Mais pas les autres, à cause de dépendance séquentielle

```
acc = (acc OP data[i]) OP data[i+1];
```

# Loop Unrolling avec réassociation

```
/* 2 x 1r loop unrolling */
void combine6(vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = acc OP (data[i] OP data[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

```
acc = (acc OP data[i]) OP data[i+1];
```

# Effet de réassociation

| Method | Integer | | Double | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| combine5 (2 x 1 unrolling) | 1.01 | 3.01 | 3.01 | 5.01 |
| combine6 (2 x 1r unrolling) | 1.01 | 1.51 | 1.51 | 2.51 |
| Latency Bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput Bound | 0.50 | 1.00 | 1.00 | 0.50 |

- **Presque 2 fois plus rapide pour Int *, FP +, FP * !**
  - Raison : casser la dépendance séquentielle
  - Pourquoi ?

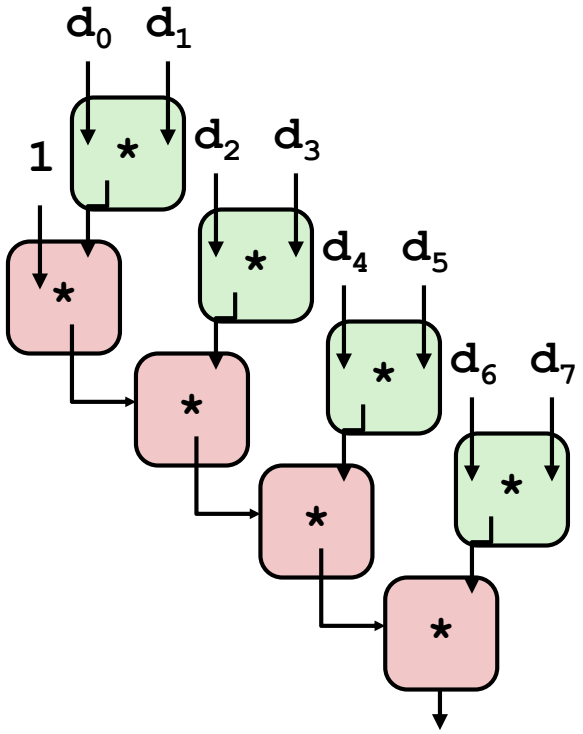
```
acc = acc OP (data[i] OP data[i+1]);
```

  - Mais attention avec l'associativité des FP (résultat incorrect ?)

# Flot de données (combine6)

```
/* 2 x 1r loop unrolling */
void combine6(vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = acc OP (data[i] OP data[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```



# Loop Unrolling avec des multiples accumulateurs

```
/* 2 x 2 loop unrolling */
void combine7(vec_ptr v, data_t *dest){
    long i;
    long length = vec_length(v);
    long limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc0 = acc0 OP data[i];
        acc1 = acc1 OP data[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

# Effet des accumulateurs multiples

| Method | Integer | | Double | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| combine5 (2 x 1 unrolling) | 1.01 | 3.01 | 3.01 | 5.01 |
| combine6 (2 x 1r unrolling) | 1.01 | 1.51 | 1.51 | 2.51 |
| combine7 (2 x 2 unrolling) | 0.81 | 1.51 | 1.51 | 2.51 |
| Latency Bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput Bound | 0.50 | 1.00 | 1.00 | 0.50 |

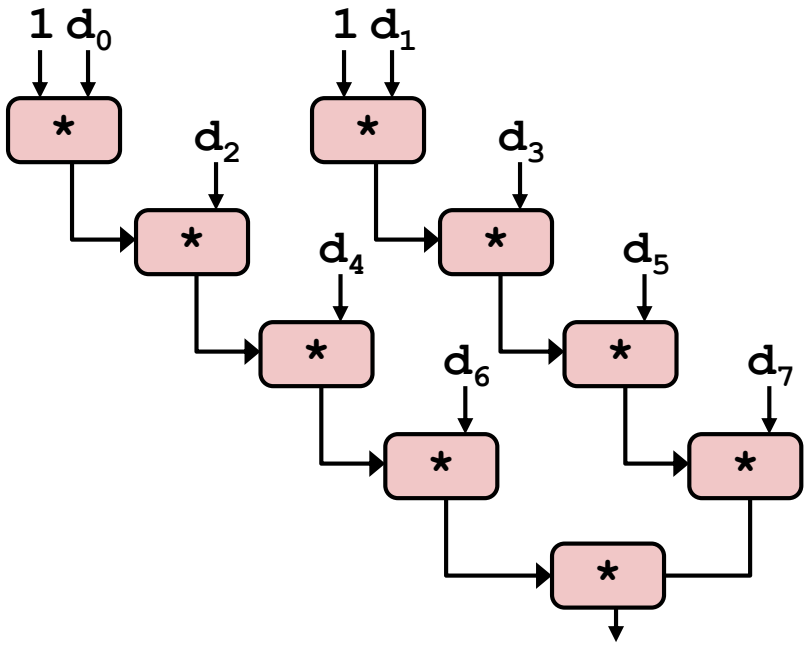
- **Presque 2 fois plus rapide pour Int *, FP +, FP * !**
  - Raison : casser la dépendance séquentielle de façon plus « propre »

```
acc0 = acc0 OP data[i];
acc1 = acc1 OP data[i+1];
```

# Flot de données (combine7)

- 2 “streams” indépendants d’opérations

```
acc0 = acc0 OP data[i];
acc1 = acc1 OP data[i+1];
```

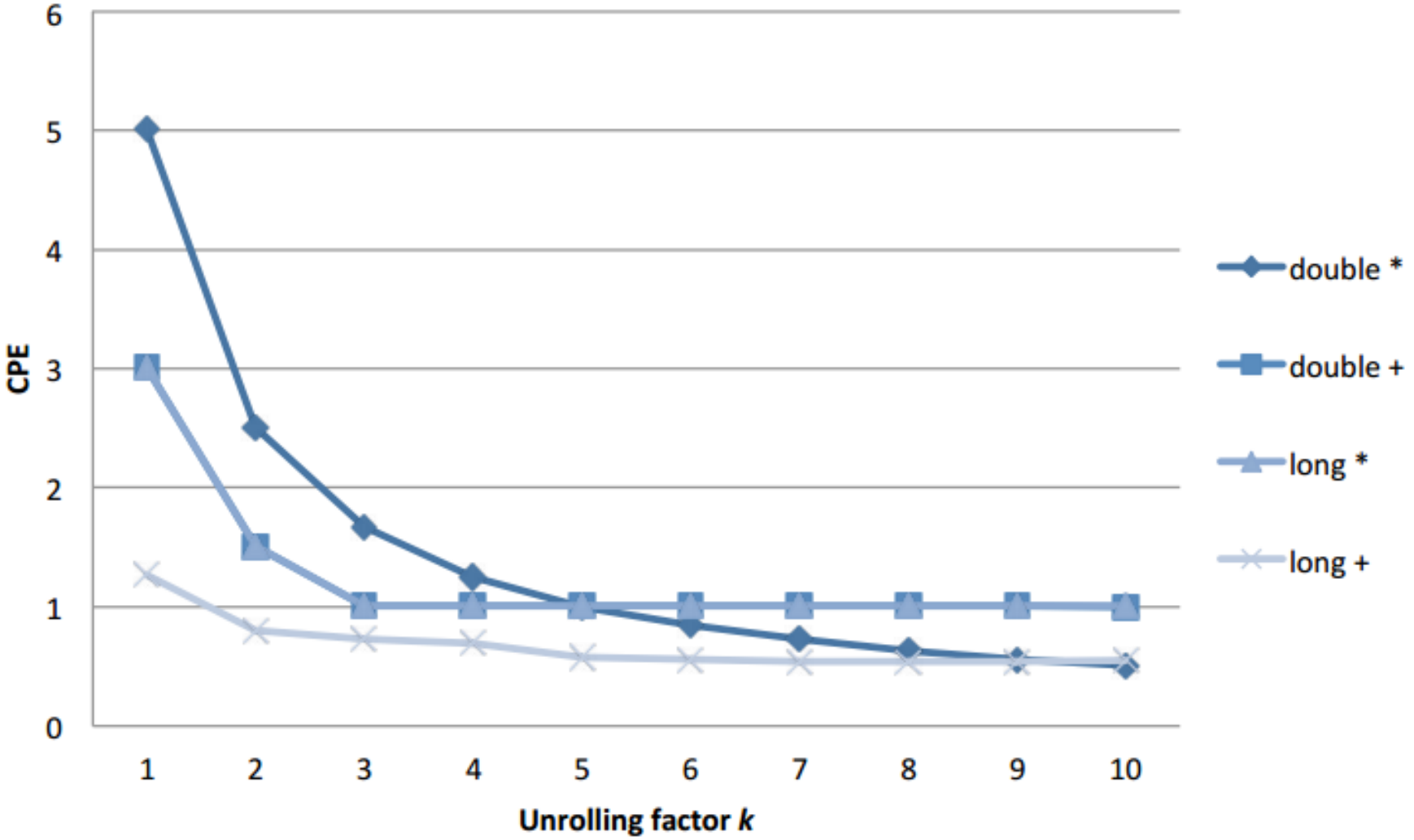


# Unrolling & Accumulation

- **Idée**
  - « Dérouler » une boucle à n'importe quel dégrée k
  - Et accumuler k valeurs en parallèle
  - k x k loop unrolling

- **Limitations**
  - Ne peut pas dépasser la limite de débit (throughput  bound) des unités d'exécution
  - Pour profiter « fully pipelining » : k >= C.L
    - C : capacity, L : latence
  - Attention à la non-associativité des nombres flottants (arrondissement ou overflow) !

# k x k loop unrolling



# Performance

| Method | Integer | | Double FP | |
|---|---|---|---|---|
| Operation | Add | Mult | Add | Mult |
| combine1 unoptimized | 22.68 | 20.02 | 19.98 | 20.18 |
| combine1 –O1 | 10.12 | 10.12 | 10.17 | 11.14 |
| combine4 | 1.27 | 3.01 | 3.01 | 5.01 |
| combine5 (2 x 1 unrolling) | 1.01 | 3.01 | 3.01 | 5.01 |
| combine6 (2 x 1r unrolling) | 1.01 | 1.51 | 1.51 | 2.51 |
| combine7 (2 x 2 unrolling) | 0.81 | 1.51 | 1.51 | 2.51 |
| combine8 (10 x 10 unrolling) | 0.55 | 1.00 | 1.01 | 0.52 |
| Latency Bound | 1.00 | 3.00 | 3.00 | 5.00 |
| Throughput Bound | 0.50 | 1.00 | 1.00 | 0.50 |

- 40 fois plus rapide que le code original !
- On peut encore faire mieux avec les instructions de vecteur (AVX) !

# Facteurs de limitation de performance

- **Chemin critique de flot de données (Σ latence)**
- **Débit (throughput bound) : I/C**
  - I : temps d'issue
  - C : nombre d'unités fonctionnelles
- **Nombre de registres : utilisation de pile quand k est grande => ralentissement !**
- **Prédiction de branche et pénalty**
  - Les processeurs modernes sont très bons pour identifier les patterns réguliers (combine2 avec bound checks => combine3)
  - Les programmeurs peuvent écrire du code du style fonctionnel pour faciliter les instructions de déplacement conditionnel (conditional move instructions)

# Exemple

```
/* Rearrange 2 vectors so that for each i, b[i] >= a[i]*/
void minmax1(long a[], long b[], long n){
    long i;
    for (i = 0; i < n; i++) {
       if (a[i] > b[i]) {
          long t = a[i];
          a[i] = b[i];
          b[i] = t;
       }
    }
}
```

- **CPE**
  - 13.5 pour les données aléatoires
  - 2.5-3.5 pour les données prévisibles

# Exemple

```
/* Rearrange 2 vectors so that for each i, b[i] >= a[i]*/
void minmax2(long a[], long b[], long n){
    long i;
    for (i = 0; i < n; i++) {
       long min = a[i] < b[i] ? a[i] : b[i];
       long max = a[i] < b[i] ? b[i] : a[i];
       a[i] = min;
       b[i] = max;
    }
}
```

- **CPE :**
  - 4
  - gcc génère des déplacements conditionnels pour ce code => moins de pénalty pour les erreurs de prédiction.

# Avoir une bonne performance

- **Bon compilateur + options**
- **Ne pas faire des bêtises ☺**
  - Vérifier des inefficacités algorithmiques
  - Écrire « compiler-friendly code »
    - Éviter des appels excessifs de procédure
    - Éviter des références mémoires inutiles :
      - Ajouter des variables temporaires
      - Stocker le résultat aux tableaux + variables globales au dernier
  - Vérifier bien la boucle à l'intérieur
- **Optimiser en fonction de machine cible**
  - Exploiter le parallélisme au niveau d'instruction
    - Loop unrolling, accumulateurs multiples, réassociation
  - Éviter des branches non-prévisibles
    - Réécrire les opérations conditionnelles
  - Écrire « cache-friendly code »

# Profiling

1. **gcc -Og –pg prog.c –o prog**
2. **./prog file.txt**
3. **gprof prog**

| % time | cum. seconds | self seconds | calls | …. | name |
|---|---|---|---|---|---|
| 97.58 | | 203.66 | 1 | | sort_word |
| 2.32 | | 4.58 | 965027 | | find_ele_rec |

http://www.thegeekstuff.com/2012/08/gprof-tutorial/