UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie informatique

RAPPORT APP3

Architecture des ordinateurs
GIF310

Présenté à
Équipe de formateurs de la session S4

Présenté par
Raphael Bouchard – bour0703
Alexis Guérard – guea0902

Sherbrooke – 20 février 2024

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

1. PERFORMANCE D'ORGANISATION

1.1 DOCUMENTATION DU CODE DE RÉFÉRENCE

Le code de référence consiste à effectuer une multiplication de matrices. L'exemple dans le code présente la multiplication d'une matrice 4x4 par une matrice 1x4. Pour réaliser une multiplication matricielle, on procède à une somme des produits des lignes de la première matrice avec les colonnes de la deuxième. Ainsi, on peut expliquer le calcul matriciel ainsi que le code par cette équation algébrique :

$$y[i] = \sum_{j=0}^{N-1} mat\_A[i \times N + j] \times x[j]$$

Pour chaque valeur de la matrice résultante de la multiplication, on applique l'équation mentionnée ci-dessus. Cela implique de multiplier la première valeur de la colonne de la première matrice par la première valeur de la ligne de la deuxième matrice correspondant à la colonne de la première matrice, et ainsi de suite. L'illustration ci-dessous aide à visualiser ce qui est expliqué dans ce paragraphe.

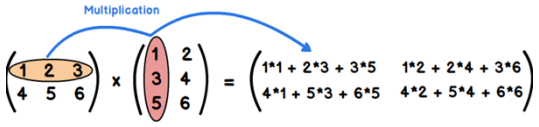


Figure 1 : Explication simplifié de la multiplication matricielle [1]

Dans la somme, nous effectuons une multiplication de $i \times N$ car dans notre code, nos matrices ne sont pas des tableaux mais des vecteurs. Cette opération équivaut donc à passer à la colonne suivante dans une matrice. Nous effectuons donc cette opération pour chaque valeur de la matrice résultante.

1.2 UNICYCLE

En comprenant mieux le fonctionnement du code de référence, on peut maintenant analyser le temps d'exécution en cycles d'horloge pour l'organisation unicycle. Pour ce faire, nous devons commencer par déterminer le nombre d'instructions effectuées dans le code assembleur MIPS. Il est essentiel de porter une attention particulière aux boucles, car elles sont exécutées plusieurs fois. Nous avons donc deux instructions au début de la fonction main pour initialiser les variables i et N. En utilisant l'application MARS, nous constatons que l'instruction lw représente en fait trois instructions. La boucle interne contient alors 16 instructions, répétées quatre fois. Nous répétons également la première instruction pour sortir de la boucle.

En continuant avec l'application MARS, nous découvrons que l'instruction sw représente également trois instructions. La boucle externe comprend neuf instructions, auxquelles s'ajoutent les 65 instructions de la boucle interne. Nous répétons également la première instruction de la boucle interne pour mettre fin à la boucle. Ainsi, nous obtenons deux instructions à la fin de la fonction main, ce qui donne un total de 301 instructions. Puisqu'une instruction prend un cycle d'horloge, le temps d'exécution de ce code est donc de 301 cycles d'horloge. Une démarche claire se retrouve dans l'annexe A (Figure 2)

En se basant sur la vitesse d'opération de 25 ns pour l'organisation unicycle, on trouve que le temps d'exécution est de 7,525 µs.

$$temps\ d'exécution = 301\ instructions\ \times 25\ ns = 7{,}525\ \mu s$$

1.3 PIPELINE BRANCHEMENT 4E ÉTAGE

Pour le pipeline avec un branchement 4e étage, il faut penser que les instructions se font en escalier. Il faut donc s'assurer que tous les registres soient à jour avant de commencer une instruction qui les utilise. Il faut ajouter des nops.

On a donc 455 pénalités pour l'organisation pipeline avec branchement au 4e étage. En ajoutant ce résultat au nombre d'instructions, on obtient un temps d'exécution de 756 cycles d'horloge. En se basant sur la vitesse d'exécution de 10 ns pour l'organisation pipeline, on trouve que le temps d'exécution est de 7,56 µs. Une démarche claire se retrouve dans l'annexe A (Figure 3).

$$temps\ d'exécution = (301 + 455)\ \times 10\ ns = 7{,}56\ \mu s$$

1.4 PIPELINE BRANCHEMENT 2E ÉTAGE

Dans le cas du pipeline avec un branchement au 2e étage, ainsi qu'avec l'unité d'avancement et l'unité de détection des aléas, il est nécessaire d'ajouter moins de nops en raison de ces modules.

On a donc 55 pénalités pour l'organisation pipeline avec branchement au 2e étage. En ajoutant ce résultat au nombre d'instructions, on obtient un temps d'exécution de 356 cycles d'horloge. En se basant sur la vitesse d'exécution de 10 ns pour l'organisation pipeline, on trouve que le temps d'exécution est de 3,56 µs. Une démarche claire se retrouve dans l'annexe A (Figure 4).

$$temps\ d'exécution = (301 + 55)\ \times 10\ ns = 3{,}56\ \mu s$$

2. PERFORMANCE SIMD

Pour optimiser le code de référence en utilisant SIMD, une extension SIMD de taille 4 sera mise en œuvre. Cela signifie que la matrice résultante sera gérée par blocs de 4 valeurs de 32 bits [2]. L'objectif est de réaliser la boucle interne en effectuant les calculs pour les 4 lignes simultanément, réduisant ainsi le nombre de boucles nécessaires. Pour cela, des extensions SIMD seront utilisées pour exécuter les mêmes instructions sur plusieurs données simultanément.

Ainsi, quelques instructions seront converties en instructions SIMD pour permettre la multiplication matricielle avec une seule boucle. Le code modifié utilisera notamment lwv au lieu de lw pour charger le registre vectoriel au lieu de charger chaque registre individuellement. De même, $addv$ sera utilisé au lieu de add pour effectuer une addition vectorielle, et $mulv$ pour une multiplication vectorielle. Enfin, swv sera utilisé pour écrire le contenu du registre vectoriel.

Avec ces extensions, le code peut fonctionner avec une seule boucle, car chaque valeur de la matrice finale sera calculée simultanément. Avec la modification du code, le nombre total d'instructions peut être calculé. Ainsi, nous avons 11 instructions pour l'initialisation des valeurs, 61 pour la boucle, et 2 pour la fin du programme.

On peut donc ensuite comparer ce temps d'exécution avec celui du code de référence de base unicycle. Avec un ratio ($\frac{301\ instructions}{74\ instructions}$), on se rend compte que cette modification du code est 4,07 fois plus rapide. Puisque les requis étaient d'utiliser 3 fois moins d'instructions, on peut anticiper qu'il est possible d'atteindre les objectifs du projet. Une démarche claire se retrouve dans l'annexe A (Figure 5).

3. PERFORMANCE DES MÉMOIRES SUR PROCESSEUR UNICYCLE

3.1 DRAM

Pour calculer le nombre de cycles d'horloge supplémentaires dus aux accès à la DRAM, nous pouvons utiliser les informations fournies. Le coût d'accès à la DRAM est de 10 cycles d'horloge. [2] Le nombre total d'accès à la DRAM peut être calculé en comptant le nombre d'instructions de chargement (lw) et de stockage (sw) dans le code. Dans la boucle interne, il y a 2 instructions lw par itération, et la boucle externe est répétée 4 fois. De plus, il y a une instruction sw à la fin de la boucle interne, répétée 4 fois par la boucle externe.

$$AccesDRAMtot = ((NInstrucLW * NBoucExt) * NBoucInt) + (NInstrucSW * NBoucExt)$$

$$AccesDRAMtot = (2 * 4 * 4) + (1 * 4) = 36$$

$$NCycleSuppl = AccesDRAMtot * Coût AccesDRAM$$

$$NCycleSuppl = 36 * 10 = 360\ cycles\ d'horloge\ supplémentaire$$

En ce qui concerne la pénalité d'accès aux instructions, étant donné qu'il s'agit d'une mémoire vive dynamique (DRAM), la pénalité d'accès aux instructions est de 10 cycles d'horloge par instruction. Comme expliqué dans la section 1.2, en ce qui concerne les performances de l'organisation unicycle, le code de référence comprend 301 instructions. Par conséquent, la pénalité totale d'accès aux instructions sera de 3010 cycle d'horloge en DRAM.

$$PAPI = Pénalité\ d'Accès\ Par\ Instruction = 10\ cycles\ d'horloge$$

$$Pénalité\ d'accès = NInstructions * PAPI = 301 * 10 = 3010\ cycles\ d'horloge$$

3.2 DRAM + CACHE

Pour calculer le nombre de coups d'horloge supplémentaires dus aux données d'une mémoire DRAM + Cache il faut tout d'abord savoir que la cache à une configuration de 256 blocs ayant 2 mots par bloc de 32 bits. Étant donné que la mémoire est DRAM, le coût d'accès à la mémoire est de 10 cycles d'horloge supplémentaire. Donc lorsque la cache obtiendra un $miss$ il y aura 10 cycles d'horloge supplémentaire plus un cycle d'horloge pour traiter la donnée. Aussi, lorsque la cache obtiendra un hit, le coup d'horloge supplémentaire pour chercher dans la cache sera de 1. De

plus, la cache écrit dans la DRAM en *write-through* ce qui veut dire qu'à chaque fois qu'une donnée est écrite dans la cache elle sera aussi écrite dans la DRAM causant un autre 10 cycles d'horloge supplémentaire. Donc, pour une écriture si on obtient un hit le nombre de coup d'horloge supplémentaire sera de 11 et si on obtient un $miss$ ça sera de 21 cycles d'horloge supplémentaire.

$$AccesDRAMMissCache = (NLWMiss\ + NSWMiss * CoûtAccesWrite) * CoûtAccesDRAM$$

$$AccesDRAMMissCache = NLWMiss * CoûtAccesDRAM + NSWMiss * (CoûtAccesWrite + CoûtAccesDRAM)$$

$$AccesDRAMMissCache = (10 * 11 + 2 * (10 + 11)) = 152\ cycles\ d'horloge\ supplémentaire$$

$$AccesDRAMHitCache = NLWHit * CoûtAccesCache + NSWHit * (CoûtAccesWrite + CoûtAccesCache)$$

$$AccesDRAMHitCache = 22 * 1 + 2 * (10 + 1) = 44\ cycles\ d'horloge\ supplémentaire$$

$$AccesDRAMtot = AccesDRAMMissCache + AccesDRAMHitCache$$

$$AccesDRAMtot = 152 + 44 = 196\ cycles\ d'horloge\ supplémentaire$$

En ce qui concerne la pénalité d'accès aux instructions, étant donné qu'il s'agit d'une mémoire DRAM avec cache, la pénalité d'accès aux instructions lorsque la cache obtiendra un $miss$ sera de 10 cycles d'horloge plus un cycle d'horloge pour la recherche dans la cache. Aussi, lorsque celle-ci obtiendra un hit la pénalité sera de 1 cycle d'horloge. Comme expliqué dans la section 1.2, en ce qui concerne les performances de l'organisation unicycle, le code de référence comprend 301 instructions. Il y a 29 instructions distinctes qui génèreront 15 $miss$ pour la cache, donc 286 $(301 - 15)$ instructions génèreront un hit pour la cache. Par conséquent, la pénalité totale d'accès aux instructions sera de 418 cycle d'horloge en DRAM.

$$PAPIMiss = Pénalité\ d'Accès\ Par\ Instruction\ Miss\ +\ Recherche\ Cache\ = 11\ cycles\ d'horloge$$

$$PAPIHit = Pénalité\ d'Accès\ Par\ Instruction\ Hit = 1\ cycle\ d'horloge$$

$$PenAccesMiss = NInstructionsMiss * PAPIMiss = 15 * 11 = 165\ cycles\ d'horloge$$

$$PenAccesHit = NInstructionsHit * PAPIHit = 286 * 1 = 286\ cycles\ d'horloge$$

$$PenAccesTot = PenAccesMiss + PenAccesHit = 165 + 286 = 451\ cycles\ d'horloge$$

4. CONFIGURATION DES CACHES

Dans l'optique d'optimiser le programme en intégrant une cache, nous avons choisi d'organiser les données en blocs de 128 bits (soit 4 mots de 32 bits chacun).

Nous avons opté pour une cache associative, permettant une gestion flexible des blocs mémoire. Lorsqu'un $miss$ se produit et que la cache est pleine, la politique adoptée est de remplacer le dernier bloc mémoire. Au lancement du programme, la cache est réinitialisée et la première tentative d'accès au premier mot de vec_entree entraîne un $miss$. Les quatre mots correspondants sont chargés dans la cache et restent accessibles tout au long de l'exécution du programme.

Ensuite, le programme tente d'accéder au premier mot de mat_entree, provoquant un nouveau $miss$ (2 au total). Une tentative similaire est faite avec vec_sortie, générant un autre $miss$ (3 au total). Les quatre mots correspondants restent accessibles dans la cache jusqu'à la fin du programme. Cependant, les données de mat_entree, changeant à chaque itération de la boucle externe, sont constamment écrasées dans le quatrième bloc de la cache (6 miss au total).

En résumé, les informations de vec_entree restent toujours disponibles dans le premier bloc de la cache, tandis que celles de vec_sortie demeurent dans le troisième bloc. En revanche, les données de mat_entree sont constamment écrasées dans le quatrième bloc de la cache, générant une pénalité de 60 cycles d'horloge tout au long du programme.

Tableau 1 : Représentation de la cache

Configuration de la cache	
Spécification	**Valeur**
Taille de la cache (Octets)	64
Taille des blocs (Octets)	16
Nombre de bloc	4
Nombre de mot par bloc	4
Taille d'un mot (bits)	32
Type de cache	Associative

5. INTÉGRATION

En partant d'une organisation unicycle avec mémoire DRAM, le système optimal à privilégier serait celui utilisant des instructions spéciales SIMD. L'emploi d'instructions vectorielles permet de réduire considérablement le nombre d'instructions nécessaires, entraînant ainsi une diminution du temps d'exécution. Nous sommes conscients des modifications nécessaires au niveau du logiciel et du matériel, mais nous croyons que les performances obtenues surpassent largement le temps investi dans leur mise en place.

L'utilisation des accès mémoires avec une cache de 4 blocs de 128 bits, ajoutée aux instructions SIMD, permettrait également des accélérations optimales. Si un changement devait être abandonné en cas de pression sur le calendrier de développement, ce serait ce changement, car il demande beaucoup de temps de développement par rapport aux gains de performance qu'il apporte.

Une autre solution, bien que fournissant de bons résultats, est trop coûteuse en temps et est donc exclue de la planification : il s'agit de l'utilisation de la mémoire SRAM. Malgré ses performances exceptionnelles, son temps de mise en place est élevé. L'utilisation d'une architecture pipeline permet une vitesse d'horloge plus élevée que celle d'une architecture unicycle, notamment 10 ns au lieu de 25 ns. L'organisation pipeline au 2^e étage avec l'unité d'avancement et l'unité de détection des aléas réduit de façon considérable le temps d'exécution. Ce serait une autre bonne alternative, mais elle est coûteuse en termes de temps, en raison des unités à implémenter.

6. RÉFÉRENCES

[1] «Data Analytics Post,» 10 Novembre 2022. [En ligne]. Available: https://dataanalyticspost.com/alphatensor-deepmind-calcul-matriciel/. [Accès le 17 Février 2024].

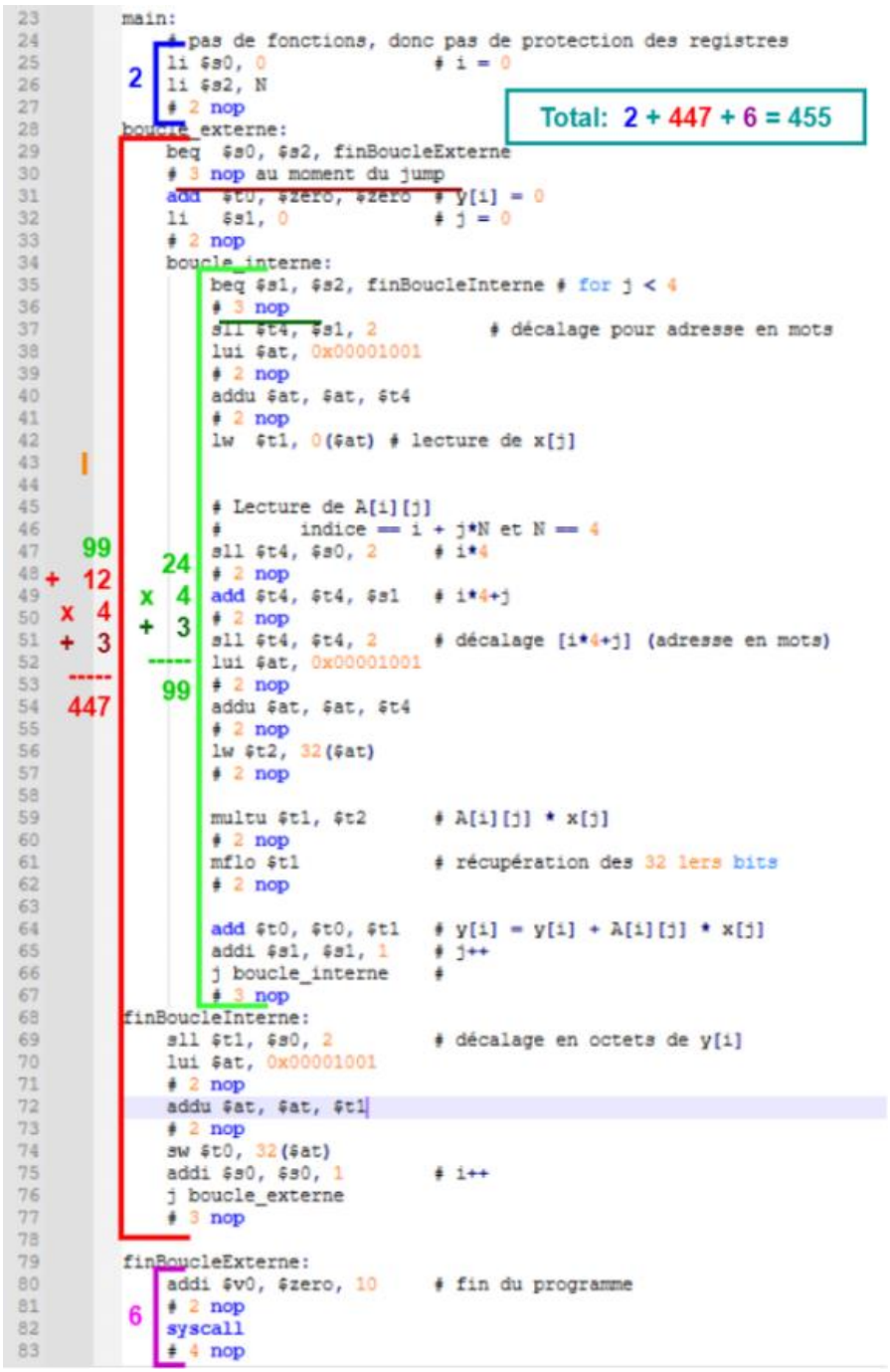
[2] Architecture et organisation des ordinateurs - Guide de l'étudiante et de l'étudiant, Sherbooke, Hiver 2024.

Annexe A FIGURES DES CALCULS

```
23          main:
24              # pas de fonctions, donc pas de protection des registres
25            2 li $s0, 0               # i = 0
26              li $s2, N
27
28          boucle_externe:
29              beq  $s0, $s2, finBoucleExterne
30              add  $t0, $zero, $zero  # y[i] = 0
31              li   $s1, 0             # j = 0
32
33              boucle_interne:
34                  beq $s1, $s2, finBoucleInterne # for j < 4
35                  sll $t4, $s1, 2           # décalage pour adresse en mots
36                  lw  $t1, vec_entree($t4) # lecture de x[j] 3 instructions
37
38                  # Lecture de A[i][j]
39            9     #       indice == i + j*N et N == 4
40          + 65    sll $t4, $s0, 2     # i*4
41          x  4    add $t4, $t4, $s1   # i*4+j
42          +  1    sll $t4, $t4, 2     # décalage [i*4+j] (adresse en mots)
43          -----   lw  $t2, mat_A($t4) # lecture de A[i][j] 3 instructions
44          297
45                  multu $t1, $t2      # A[i][j] * x[j]
46                  mflo $t1            # récupération des 32 1ers bits
47
48                  add $t0, $t0, $t1   # y[i] = y[i] + A[i][j] * x[j]
49                  addi $s1, $s1, 1    # j++
50                  j boucle_interne    #
51
52          finBoucleInterne:
53              sll $t1, $s0, 2         # décalage en octets de y[i]
54              sw $t0, vec_sortie($t1) # écriture de y[i]  3 instructions
55              addi $s0, $s0, 1        # i++
56              j boucle_externe
57
58          finBoucleExterne:
59            2 addi $v0, $zero, 10     # fin du programme
60              syscall
61
```

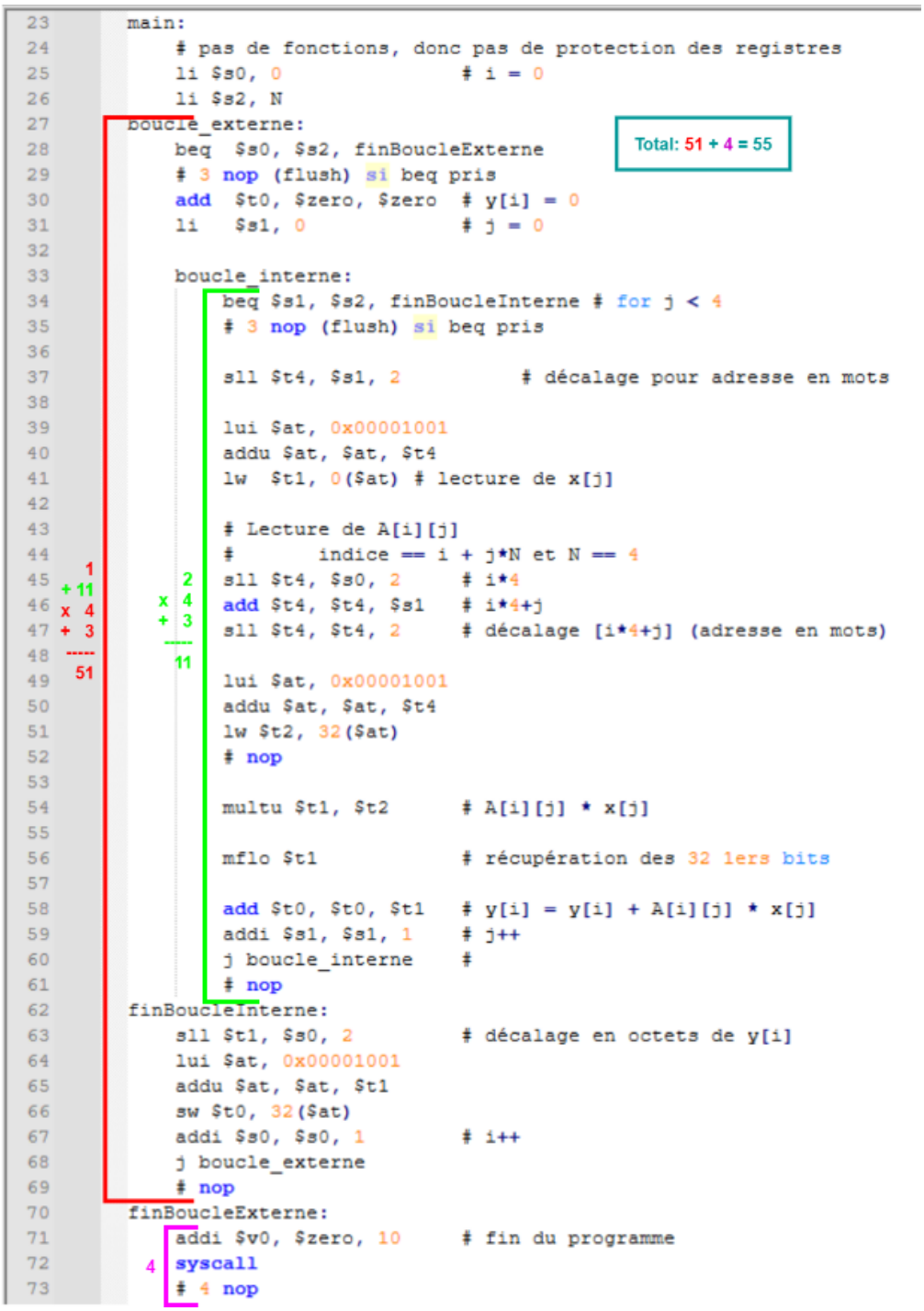
Total : 2 + 297 + 2 = 301 instructions

Figure 2 : Calcul du nombre d'instructions dans le code de référence




```
23      main:
24          # pas de fonctions, donc pas de protection des registres
25          li $s0, 0               # i = 0
26          li $s2, N
27          # 2 nop
28      boucle_externe:
29          beq  $s0, $s2, finBoucleExterne
30          # 3 nop au moment du jump
31          add  $t0, $zero, $zero  # y[i] = 0
32          li   $s1, 0             # j = 0
33          # 2 nop
34          boucle_interne:
35              beq $s1, $s2, finBoucleInterne # for j < 4
36              # 3 nop
37              sll $t4, $s1, 2          # décalage pour adresse en mots
38              lui $at, 0x00001001
39              # 2 nop
40              addu $at, $at, $t4
41              # 2 nop
42              lw  $t1, 0($at) # lecture de x[j]
43
44
45              # Lecture de A[i][j]
46              #       indice == i + j*N et N == 4
47              sll $t4, $s0, 2     # i*4
48              # 2 nop
49              add $t4, $t4, $s1   # i*4+j
50              # 2 nop
51              sll $t4, $t4, 2     # décalage [i*4+j] (adresse en mots)
52              lui $at, 0x00001001
53              # 2 nop
54              addu $at, $at, $t4
55              # 2 nop
56              lw $t2, 32($at)
57              # 2 nop
58
59              multu $t1, $t2      # A[i][j] * x[j]
60              # 2 nop
61              mflo $t1            # récupération des 32 lers bits
62              # 2 nop
63
64              add $t0, $t0, $t1   # y[i] = y[i] + A[i][j] * x[j]
65              addi $s1, $s1, 1    # j++
66              j boucle_interne    #
67              # 3 nop
68      finBoucleInterne:
69          sll $t1, $s0, 2         # décalage en octets de y[i]
70          lui $at, 0x00001001
71          # 2 nop
72          addu $at, $at, $t1
73          # 2 nop
74          sw $t0, 32($at)
75          addi $s0, $s0, 1        # i++
76          j boucle_externe
77          # 3 nop
78
79      finBoucleExterne:
80          addi $v0, $zero, 10     # fin du programme
81          # 2 nop
82          syscall
83          # 4 nop
```

Figure 3 : Pénalité de retard pour Pipeline branchement 4e étage




```
23      main:
24          # pas de fonctions, donc pas de protection des registres
25          li $s0, 0               # i = 0
26          li $s2, N
27      boucle_externe:
28          beq  $s0, $s2, finBoucleExterne
29          # 3 nop (flush) si beq pris
30          add  $t0, $zero, $zero  # y[i] = 0
31          li   $s1, 0             # j = 0
32
33          boucle_interne:
34              beq $s1, $s2, finBoucleInterne # for j < 4
35              # 3 nop (flush) si beq pris
36
37              sll $t4, $s1, 2          # décalage pour adresse en mots
38
39              lui $at, 0x00001001
40              addu $at, $at, $t4
41              lw  $t1, 0($at) # lecture de x[j]
42
43              # Lecture de A[i][j]
44              #       indice == i + j*N et N == 4
45      1       sll $t4, $s0, 2      # i*4
46    + 11    x 4
47    x 4     + 3
48    + 3     ----
49    ----     11
50    51        lui $at, 0x00001001
51              addu $at, $at, $t4
52              lw $t2, 32($at)
53              # nop
54
55              multu $t1, $t2       # A[i][j] * x[j]
56
57              mflo $t1             # récupération des 32 1ers bits
58
59              add $t0, $t0, $t1    # y[i] = y[i] + A[i][j] * x[j]
60              addi $s1, $s1, 1     # j++
61              j boucle_interne     #
62              # nop
63      finBoucleInterne:
64          sll $t1, $s0, 2          # décalage en octets de y[i]
65          lui $at, 0x00001001
66          addu $at, $at, $t1
67          sw $t0, 32($at)
68          addi $s0, $s0, 1         # i++
69          j boucle_externe
70          # nop
71      finBoucleExterne:
72          addi $v0, $zero, 10      # fin du programme
73      4   syscall
            # 4 nop
```

Figure 4 : Pénalité de retard pour Pipeline branchement 2e étage

```
24  main:
25      # pas de fonctions, donc pas de protection des registres
26      li $s0, 0               # i = 0
27      li $s1, N
28  11
29      lwv $s2, vec_entree
30      lwv $s3, vec_sortie
31      lwv $s5, vec_temp
32
33  boucle_externe:
34      beq  $s0, $s1, finBoucleExterne
35
36      # Chargement d'une ligne de la matrice A
37      sll $t4, $s0, 2
38      lwv $s4, mat_A($t4)
39
40
41      # Multiplication vectorielle
42      mulv $t4($s5), $s2, $s4
43
44      # Addition vectorielle
45      addv $s3, $s3, $s5
46
47      # Stockage de la ligne de Y
48      swv $s3, vec_sortie($t4)
49
50      sw $zero, $t4($s5)
51
52      # Incrémentation du compteur de ligne
53      addi $s0, $s0, 1
54
55      j boucle_externe
56
57  finBoucleExterne:
58      addi $v0, $zero, 10     # fin du programme
59  2   syscall
60
```

15 x 4 + 1 ---- 61

Total : 11 + 61 + 2 = 74

Figure 5 : Calcul du nombre d'instruction SIMD