



**POLYTECHNIQUE
MONTRÉAL**

UNIVERSITÉ
D'INGÉNIERIE

INF3610 – Systèmes embarqués

Automne 2024

TP No. 2

Groupe 02



Soumis à



Lundi 25 Novembre 2024

Table des matières	2
Explication de la démarche utilisé	3
Question 1	16
Question 2	20
Question 3	21
Question 4	23
Q1S	24
Q2S	25
Q3S	28

Étape 1 : Synthèse HLS et optimisation

Explication de la démarche utilisé ainsi que les captures d'écrans:

- Solution 1.1:

1. Démarche :
 - Exécution du code initial sans directives.
 - Analyse des ressources utilisées et de la latence.
2. Résultats :

Summary

Latency		Interval		
min	max	min	max	Type
818581	818581	818581	818581	none

Detail

Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- L1	818580	818580	19490	-	-	42	no
+ L2	19488	19488	464	-	-	42	no
++ L3	462	462	11	-	-	42	no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	143
FIFO	-	-	-	-
Instance	-	5	348	711
Memory	-	-	-	-
Multiplexer	-	-	-	116
Register	-	-	239	-
Total	0	5	587	970
Available	280	220	106400	53200
Utilization (%)	0	2	~0	1

- Solution 1.2:

Méthode:

Pipeliner la boucle principale (L2) :

Un pragma **#pragma HLS PIPELINE II=1** a été ajouté au-dessus de la boucle **L2** afin de réduire l'Initiation Interval (II) à **1**, permettant de lancer une nouvelle itération à chaque cycle. Cela optimise le débit en rendant le traitement plus rapide.

Partitionner les tableaux pour des accès parallèles :

Les matrices **a** et **b** ont été partitionnées à l'aide des directives :

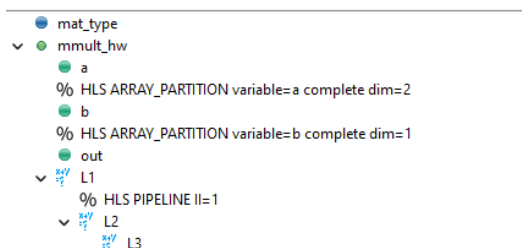
- **#pragma HLS ARRAY_PARTITION variable=a complete dim=2**
- **#pragma HLS ARRAY_PARTITION variable=b complete dim=1** Ces partitionnements permettent des accès simultanés aux données des matrices, éliminant les conflits de ressources liés à la mémoire.

Optimisation ciblée des DSPs :

Les modifications visent à utiliser efficacement les DSPs disponibles sur le FPGA. Avec cette méthode, **95% des DSPs** ont été utilisés, maximisant ainsi leur potentiel pour les calculs.

Simulation et synthèse :

Après l'ajout des directives, le code a été synthétisé pour évaluer l'utilisation des ressources et vérifier que l'II=1 était atteint. La simulation a confirmé que la fonctionnalité était correcte avec une latence totale de **1981 cycles**.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.634	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1981	1981	1981	1981	none

Detail

Instance

Loop

	Latency		Initiation Interval				
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1_L2	1979	1979	217	1	1	1764	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	83
FIFO	-	-	-	-
Instance	-	210	14616	29862
Memory	-	-	-	-
Multiplexer	-	-	-	75
Register	0	-	5964	640
Total	0	211	20580	30660
Available	280	220	106400	53200
Utilization (%)	0	95	19	57

- Solution 1.3:

Méthode :

Pipeline ajusté :

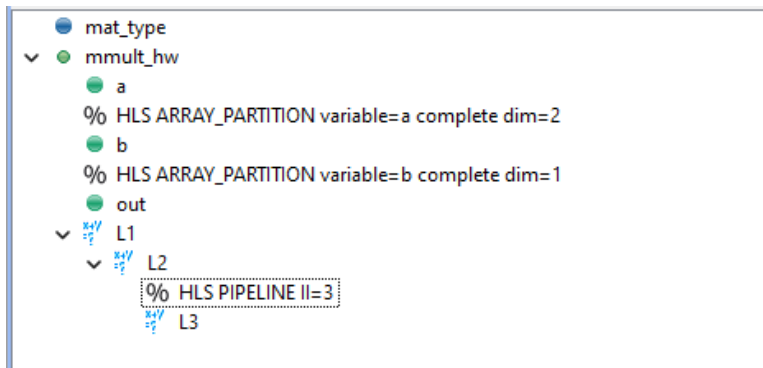
- Application du pragma **#pragma HLS PIPELINE II=3** sur la boucle L2 pour réduire la pression sur les ressources tout en maintenant un débit acceptable.

Partitionnement complet des matrices :

- Utilisation de **#pragma HLS ARRAY_PARTITION** avec l'option **complete** pour les dimensions critiques des matrices **a** et **b** afin d'optimiser les accès parallèles en mémoire.

Optimisation des ressources :

- Ajustement de l'Initiation Interval (II) et des directives de partitionnement pour limiter l'utilisation des ressources FPGA (BRAM, DSP, FF et LUT) à moins de **45%**, conformément aux spécifications.



Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	Type
5507	5507	5507	5507	none

Detail

Instance

Loop

Loop Name	Latency		Initiation Interval		Trip Count	Pipelined
	min	max	achieved	target		
- L1_L2	5505	5505	217	3	3	1764 yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	81
FIFO	-	-	-	-
Instance	-	70	4872	9954
Memory	-	-	-	-
Multiplexer	-	-	-	1272
Register	0	-	8323	2339
Total	0	71	13195	13646
Available	280	220	106400	53200
Utilization (%)	0	32	12	25

- Solution 1.4:

Méthode :

- Placement du pipeline** : HLS pipeline juste au-dessus de L2 pour optimiser les boucles L2 et L3.
- Optimisation mémoire** : Utilisation des directives HLS ARRAY_PARTITION complètes pour :
 - a (dim=2),
 - b (dim=0).
- Ajustement de DIM** : Test itératif des tailles de matrices pour atteindre 95 % des ressources (DSP, LUT, FF).

Résultat :

- Taille de la matrice : **21x21**.

```

mat_type]
mmult_hw
  a
  % HLS ARRAY_PARTITION variable=a complete dim=2
  b
  % HLS ARRAY_PARTITION variable=b complete dim=0
  out
  L1
  % HLS PIPELINE
  L2
  L3

```

Summary

Latency		Interval		
min	max	min	max	Type
342	342	342	342	none

Detail

Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- L1	340	340	121	11	1	21	yes

Utilization Estimates

Summary

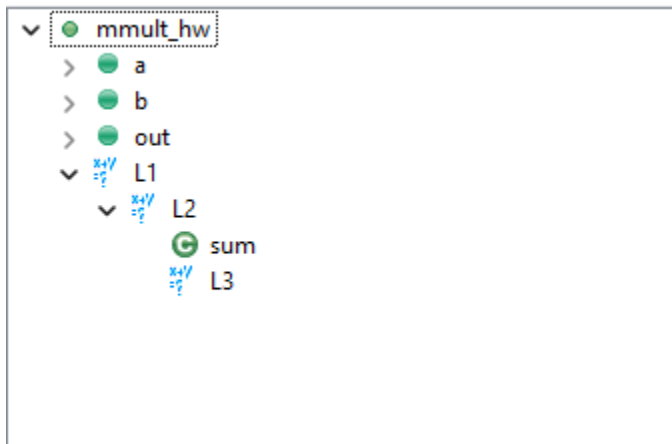
Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	343
FIFO	-	-	-	-
Instance	-	205	14268	29151
Memory	-	-	-	-
Multiplexer	-	-	-	8288
Register	0	-	47367	11520
Total	0	205	61635	49302
Available	280	220	106400	53200
Utilization (%)	0	93	57	92

- Solution 2.1:

Méthode :

1. Code de départ :
 - Aucun pragma d'optimisation ajouté.
 - Utilisation du format demi-précision pour les calculs.
2. Configuration de la matrice :
 - Dimension fixe : 42x42 (définie dans `mult.h`).
3. Synthesis :
 - Code exécuté directement sans directives supplémentaires.

Résultats obtenus :



Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
818581	818581	818581	818581	none

Detail

Instance

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- L1	818580	818580	19490	-	-	42	no
+ L2	19488	19488	464	-	-	42	no
++ L3	462	462	11	-	-	42	no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	143
FIFO	-	-	-	-
Instance	-	4	200	152
Memory	-	-	-	-
Multiplexer	-	-	-	116
Register	-	-	175	-
Total	0	4	375	411
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

- Solution 2.2:

Méthode:

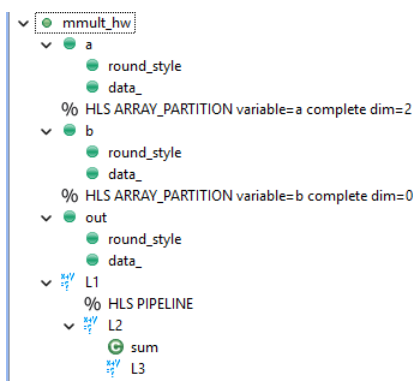
1. Approche de recherche binaire :
 - Début avec la valeur DIM = 42 (utilisée dans la solution 1.1).
 - Test successivement des valeurs intermédiaires en divisant ou augmentant la taille par étapes binaires (ex. $42 \rightarrow 21 \rightarrow 32 \rightarrow 26$).
 - Ajustement de DIM en fonction de l'utilisation des ressources FPGA (DSP, BRAM, LUT, FF).
2. Optimisations appliquées :
 - Pragma HLS PIPELINE appliqué au-dessus de L2.
 - HLS ARRAY_PARTITION utilisé pour permettre des accès parallèles en mémoire :
 - Variable a : Partition complète sur la dimension 2.
 - Variable b : Partition complète sur la dimension 0.

justification

La dimension **DIM = 26** respecte :

- Une utilisation équilibrée des ressources FPGA (<95% pour au moins un type).
- Une latence optimisée grâce à l'utilisation du pragma au-dessus de L2.

résultat:



Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
475	475	475	475	none

Detail

Instance

Loop

	Latency		Initiation Interval				
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1	473	473	149	13	1	26	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	388
FIFO	-	-	-	-
Instance	-	208	10400	7904
Memory	-	-	-	-
Multiplexer	-	-	-	11682
Register	0	-	55096	17568
Total	0	208	65496	37542
Available	280	220	106400	53200
Utilization (%)	0	94	61	70

- Solution 2.3:

Méthode utilisée:

1. Pipeline avec intervalle d'initiation ajusté

Directive utilisée :

```
#pragma HLS PIPELINE II=2
```

- Le II (Initiation Interval) de 2 a été spécifié pour la boucle L2. Cela permet de démarrer une nouvelle itération toutes les 2 cycles, réduisant ainsi la pression sur les ressources.
- Cela ajuste le débit sans sacrifier significativement les performances.

2. Partitionnement des tableaux avec facteur de blocs

Directives utilisées :

```
#pragma HLS ARRAY_PARTITION variable=a block factor=13 dim=2
```

```
#pragma HLS ARRAY_PARTITION variable=b block factor=13 dim=1
```

- Les tableaux ont été partitionnés avec un block factor de 13, divisant chaque dimension en blocs de 13 éléments.
- Cette méthode réduit le parallélisme en mémoire tout en conservant un débit suffisant pour la configuration avec II=2.

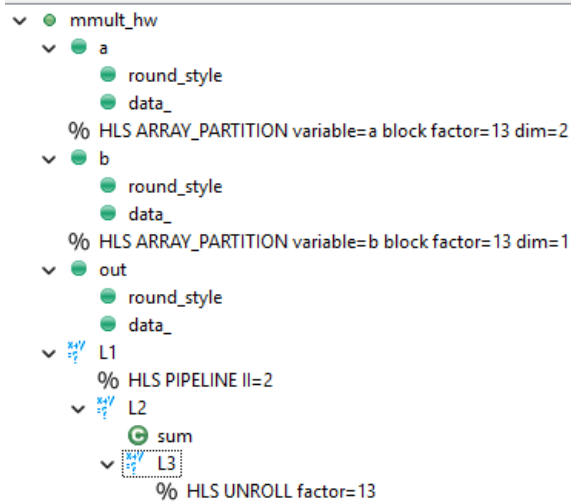
3. Déroulage partiel de la boucle interne (L3)

Directive utilisée :

`#pragma HLS UNROLL factor=13`



- La boucle L3 a été déroulée partiellement avec un facteur de 13, correspondant au facteur de bloc utilisé pour les partitions des tableaux.
- Cela permet de traiter 13 multiplications et accumulations simultanément, ce qui équilibre les performances et l'utilisation des ressources.



Summary

Latency		Interval		
min	max	min	max	Type
816	816	816	816	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1	814	814	165	26	2	26	yes

Utilization Estimates

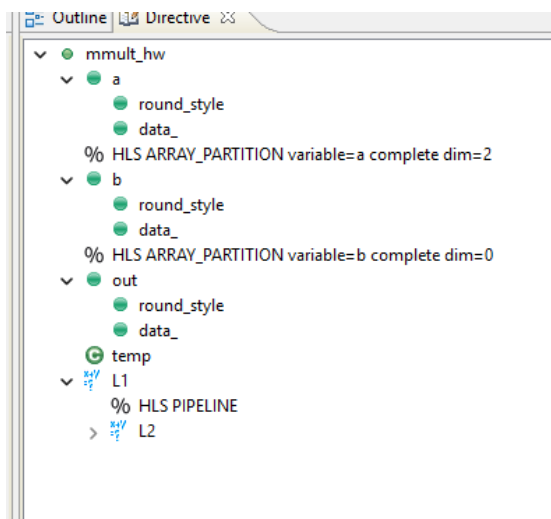
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	394
FIFO	-	-	-	-
Instance	-	104	5200	3952
Memory	-	-	-	-
Multiplexer	-	-	-	10294
Register	0	-	46629	14240
Total	0	104	51829	28880
Available	280	220	106400	53200
Utilization (%)	0	47	48	54

- Solution 3.1:

méthode:

On a rajouté les directives suivantes dans le code de la solution 3.1 en mettant 26 comme taille de matrice trouvée en 2.2.



Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
475	475	475	475	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1	473	473	149	13	1	26	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	388
FIFO	-	-	-	-
Instance	-	208	10400	7904
Memory	-	-	-	-
Multiplexer	-	-	-	11682
Register	0	-	55096	17568
Total	0	208	65496	37542
Available	280	220	106400	53200
Utilization (%)	0	94	61	70

- Solution 3.2:

```

v ● mmult_hw
  %0 HLS RESOURCE variable=sum core=HAddSub_nodsp
  > ● a
    %0 HLS ARRAY_PARTITION variable=a complete dim=2
  > ● b
    %0 HLS ARRAY_PARTITION variable=b complete dim=1
  > ● out
  ● temp
    %0 HLS RESOURCE variable=temp core=HMul_nodsp
  v L1
    v L2
      %0 HLS PIPELINE II=1
      ● sum
      L3

```

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
20167	20167	20167	20167	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1_L2	20165	20165	567	1	1	19600	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	95
FIFO	-	-	-	-
Instance	-	0	27440	49840
Memory	-	-	-	-
Multiplexer	-	-	-	75
Register	0	-	9914	640
Total	0	1	37354	50650
Available	280	220	106400	53200
Utilization (%)	0	~0	35	95

- Solution 3.3:

Summary

Latency		Interval		
min	max	min	max	Type
39767	39767	39767	39767	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1_L2	39765	39765	568	2	2	19600	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	166
FIFO	-	-	-	-
Instance	-	0	13720	24920
Memory	-	-	-	-
Multiplexer	-	-	-	6390
Register	0	-	18415	15279
Total	0	1	32135	46755
Available	280	220	106400	53200
Utilization (%)	0	~0	30	87

Étape 2 : Ajout des interfaces et production des 3 IPs

Dans cette étape, nous allons générer trois IPs en utilisant les solutions explorées dans l'étape précédente (voir Tableau 1). Ces IPs incluront des interfaces AXI Stream pour fonctionner avec le DMA, permettant ainsi une communication efficace entre le processeur ARM Cortex A9 de la carte PYNQ et le matériel FPGA.

Étape 3 : Conception du système dans Vivado

Ici on crée 3 projets Vivado dans lesquels on fera l'importation du hls_accel respectif.

Étape 4 : Conception de la partie logiciel du système avec SDK et tests de performance

Pour chaque projet Vivado déjà créé dans le code départ (projet_sol2_2, projet_sol3_3 et projet_sol4_1), on va créer un SDK.

Questions pour le rapport

Question 1

Q1.1) Que fait le testbench?

Le testbench vérifie la fonctionnalité de la multiplication matricielle en générant des entrées de test, en exécutant le code de multiplication, et en comparant les résultats de sortie avec des valeurs attendues pour valider la précision et le fonctionnement du code.

Q1.2) Combien de ressources DSP sont requises pour la solution 1.1

Ressources DSP utilisées : 5 DSP48E

Ces 5 DSPs sont utilisés pour effectuer les opérations de multiplication et d'accumulation dans le calcul matriciel, car le compilateur Vivado HLS a mappé ces opérations sur les DSPs disponibles.

Q1.3) Combien de ressources DSP sont requises pour la solution 1.2.

Pour la solution 1.2, le rapport généré après la synthèse montre les informations suivantes dans la section "**Utilization Estimates**" :

- **DSP48E utilisé : 211**

Q1.4) Combien de ressources DSP sont requises pour la solution 1.3.

Pour la solution 1.3, le rapport généré après la synthèse montre les informations suivantes dans la section "**Utilization Estimates**" :

- **DSP48E utilisé : 71**

Q1.5) Combien de ressources DSP sont requises pour la solution 1.4.

Pour la solution 1.4, le rapport généré après la synthèse montre les informations suivantes dans la section "**Utilization Estimates**" :

- **DSP48E utilisé : 205**

Q1.6) Comment déterminer le nombre de ressources DSP requises par multiplication/accumulation avec un float?

Pour déterminer le nombre de DSP requis par multiplication/accumulation avec un float, **on peut se baser sur le nombre de DSP utilisé dans la solution 1.1 sans aucune optimisation, et on conclut que ce nombre est de 5 DSPs.**

Q1.7) Comparez la complexité (avec notation O) en temps d'exécution et ressources DSP entre les solutions 1.1, 1.2, 1.3 et 1.4. Expliquez.

Solution 1.1

- Temps d'exécution: $O(N^3)$
 - Explication: Sans directives d'optimisation, la multiplication matricielle s'exécute de manière séquentielle. Chaque élément du produit matriciel nécessite une somme de NNN produits scalaires, ce qui conduit à un total de N^3 opérations. Ainsi, la complexité temporelle est proportionnelle à N^3 .
 - Utilisation des ressources DSP: $O(1)$ ou minimal
 - Explication: Sans parallélisme, le matériel réutilise une seule unité de multiplication et d'accumulation (MAC) pour tous les calculs. Par conséquent, le nombre de DSP utilisés est indépendant de la taille de la matrice N , d'où une complexité de $O(1)$.
-

Solution 1.2

- Temps d'exécution: $O(N^2)$
 - Explication: En appliquant le pipeline avec un intervalle d'initiation (II) de 1 et en partitionnant complètement les tableaux, nous parallélisons les calculs sur les dimensions critiques. Chaque itération de la boucle externe (L2) démarre à chaque cycle d'horloge, et grâce à la partition complète, les opérations de la boucle interne (L3) sont effectuées en parallèle. Cela réduit le temps d'exécution proportionnellement à N^2 .
 - Utilisation des ressources DSP: $O(N^2)$
 - Explication: Le parallélisme maximal nécessite un DSP pour chaque opération de multiplication simultanée. Comme il y a N^2 éléments dans les matrices de sortie, le nombre total de DSP utilisés est proportionnel à N^2 .
-

Solution 1.3

- Temps d'exécution: $O(N^2)$ (avec un facteur constant plus grand)
 - Explication: En augmentant l'II à 3, nous réduisons le parallélisme, ce qui augmente le temps d'exécution par rapport à la solution 1.2. Cependant, la complexité reste proportionnelle à N^2 puisque le parallélisme est toujours exploité, mais avec un débit réduit.
- Utilisation des ressources DSP: $O(N^2/k)$ où k est un facteur constant (>1)

- Explication: En réduisant le parallélisme ($II=3$), nous diminuons le nombre de DSP nécessaires. Le nombre de DSP est proportionnel à N^2/k , avec k reflétant la réduction du parallélisme. Ainsi, les ressources DSP sont réduites tout en maintenant une complexité en $O(N^2)$.

Solution 1.4

- Temps d'exécution: $O(N^2)$ avec N réduit (ici $N=21$)
 - Explication: En ajustant la taille de la matrice à 21×21 pour maximiser l'utilisation des ressources FPGA (95%), nous conservons une complexité temporelle de $O(N^2)$, mais avec un N plus petit, ce qui réduit le temps d'exécution total.
- Utilisation des ressources DSP: $O(N^2)$
 - Explication: En maximisant les ressources disponibles pour une matrice plus petite, nous utilisons efficacement les DSP. Le nombre de DSP est toujours proportionnel à N^2 , mais comme N est plus petit, le nombre total de DSP utilisés est adapté aux contraintes du FPGA.

Q1.8) À la solution 1.2, pour le pragma HLS ARRAY_PARTITION, proposez une valeur de block factor plutôt qu'un complète qui donnera un résultat similaire au niveau débit, temps d'exécution et ressources. Justifiez.

Proposition de block factor :

Pour réduire l'utilisation des ressources tout en maintenant des performances similaires, nous pouvons utiliser une partition par blocs avec un **block factor égal à la taille de la dimension partitionnée**, c'est-à-dire un **block factor de 14**. Cela signifie que nous allons partitionner les tableaux en blocs de 14 éléments.

```
#pragma HLS ARRAY_PARTITION variable=a block factor=14 dim=2
```

```
#pragma HLS ARRAY_PARTITION variable=b block factor=14 dim=1
```

Justification :

- **Accès parallèle suffisant :** En partitionnant les tableaux avec un block factor de 14, nous créons 3 blocs ($42 / 14 = 3$). Cela permet d'accéder à 14 éléments en parallèle, ce qui est un compromis entre une partition complète et aucune partition.
- **Ressources réduites :** Cette approche réduit le nombre de registres utilisés, car nous n'avons pas besoin de stocker chaque élément individuellement. Les ressources FPGA consommées sont ainsi significativement diminuées par rapport à une partition complète.
- **Maintien du débit :** En combinant cette partition par blocs avec un unrolling partiel de la boucle L3 par un facteur de 14, nous pouvons traiter 14 itérations de L3 en parallèle. Cela

permet de maintenir un II de 1 pour la boucle L2, car nous pouvons toujours amorcer une nouvelle itération à chaque cycle d'horloge.

- **Performance similaire** : Bien que nous ne puissions pas traiter les 42 éléments de L3 en parallèle, le traitement de 14 éléments à la fois réduit le nombre total de cycles nécessaires pour compléter L3. Ainsi, le temps d'exécution global reste proche de celui obtenu avec une partition complète.

Q1.9) À la solution 1.3, pour le pragma HLS ARRAY_PARTITION, proposez une valeur de block factor plutôt qu'un complète qui donnera un résultat similaire au niveau débit, temps d'exécution et ressources. Justifiez.

Proposition de block factor :

Pour atteindre les mêmes performances avec une utilisation réduite des ressources, nous pouvons proposer un **block factor de 21** pour la partition des tableaux :

```
#pragma HLS ARRAY_PARTITION variable=a block factor=21 dim=2
```

```
#pragma HLS ARRAY_PARTITION variable=b block factor=21 dim=1
```

Justification :

- **Réduction des ressources** : En augmentant le block factor à 21, nous réduisons le nombre de partitions, ce qui diminue l'utilisation des ressources FPGA par rapport à une partition complète.
- **Accès parallèle adéquat** : Avec un block factor de 21, nous pouvons accéder à 21 éléments en parallèle. En ajustant l'unrolling de la boucle L3 à un facteur de 21, nous pouvons traiter 21 itérations simultanément.
- **Correspondance avec l'II ajusté** : Étant donné que nous avons augmenté l'II à 3, nous n'avons pas besoin d'accéder à tous les éléments en parallèle. Un block factor de 21 est suffisant pour satisfaire les exigences de débit avec l'II=3.
- **Maintien du débit** : En traitant 21 éléments à la fois dans la boucle L3, nous équilibrons le compromis entre le débit et l'utilisation des ressources. La latence globale est légèrement augmentée par rapport à une partition complète, mais reste conforme aux spécifications de la solution 1.3.

Q1.10) Combien de lignes de codes VHDL demande l'implémentation de la solution 1.2?

le nombre de ligne du code vhdL généré pour la solution 1.2 est de 10037

```
10030     tmp_1_cast_fu_1753_p1 <= std_logic_vector(IEEE.numeric_std.resize(signed(grp_fu_1757_p3),64));
10031
10032     tmp_2_fu_1736_p1 <= std_logic_vector(IEEE.numeric_std.resize(unsigned(ib_mid2_fu_1715_p3),64));
10033     tmp_mid2_fu_1731_p1 <= std_logic_vector(IEEE.numeric_std.resize(unsigned(tmp_mid2_v_fu_1723_p3),64));
10034     tmp_mid2_v_fu_1723_p3 <=
10035         ia_1_fu_1703_p2 when (exitcond_fu_1709_p2(0) = '1') else
10036         ap_phi_mux_ia_phi_fu_1336_p4;
10037 end behav;
10038
```

Question 2

Q2.1) Combien de ressources DSP sont requises pour la solution 2.1?

Pour la solution 2.1, le rapport généré après la synthèse montre les informations suivantes dans la section "**Utilization Estimates**" :

- **DSP48E utilisé : 4**

Q2.2) Combien de ressources DSP sont requises pour la solution 2.2?

Pour la solution 2.2, le rapport généré après la synthèse montre les informations suivantes dans la section "**Utilization Estimates**" :

- **DSP48E utilisé : 208**

Q2.3) Combien de ressources DSP sont requises pour la solution 2.3?

Pour la solution 2.3, le rapport généré après la synthèse montre les informations suivantes dans la section "**Utilization Estimates**" :

- **DSP48E utilisé : 104**

Q2.4) Combien de ressources DSP requises par multiplication/accumulation avec un half float?

Pour déterminer le nombre de DSP requis par multiplication/accumulation avec un half float, **on peut se baser sur le nombre de DSP utilisé dans la solution 2.1 sans aucune optimisation, et on conclut que ce nombre est de 4 DSPs.**

Question 3

Q3.1) Expliquez le rôle des pragmas de ressources utilisé à la solution 3.2

Ces pragmas servent à contrôler l'implémentation matérielle des opérations arithmétiques en demi-précision (16 bits) associées aux variables sum et temp. Plus précisément :

- `#pragma HLS RESOURCE variable=sum core=HAddSub_nodsp` indique à l'outil Vivado HLS que l'opération d'addition en demi-précision liée à la variable sum doit être réalisée **sans utiliser les blocs DSP** du FPGA. L'opération sera donc implémentée en utilisant les ressources logiques programmables, c'est-à-dire les LUTs et les flip-flops (FFs).
- De même, `#pragma HLS RESOURCE variable=temp core=HMul_nodsp` force l'opération de multiplication en demi-précision associée à la variable temp à être réalisée sans recourir aux DSP, en utilisant les ressources logiques du FPGA.

Le rôle principal de ces pragmas est donc de **contrôler l'utilisation des ressources matérielles** lors de la synthèse. En évitant l'utilisation des DSP pour les opérations arithmétiques en demi-précision, on libère ces ressources pour d'autres parties du design qui pourraient en avoir besoin ou on adapte le design aux FPGA disposant d'un nombre limité de DSP. En synthétisant les opérations arithmétiques avec des LUTs et des FFs, on utilise les ressources logiques programmables, ce qui peut augmenter l'utilisation de ces ressources mais permet d'économiser les DSP.

Q3.2) Justifiez l'utilisation du code de la figure 4 pour les pragmas de ressource utilisé à la solution 3.2

Le code de la figure 4 modifie la structure des boucles de la multiplication matricielle en séparant explicitement les opérations de multiplication et d'addition. Cette séparation est essentielle pour pouvoir appliquer les pragmas de ressources de manière ciblée :

- En isolant la multiplication dans `temp = a[ia][id] * b[id][ib];`, nous pouvons appliquer le pragma `#pragma HLS RESOURCE variable=temp core=HMul_nodsp` directement à la variable temp. Cela contrôle précisément comment l'opération de multiplication en demi-précision est implémentée.
- De même, en ayant l'accumulation séparée dans `sum = sum + temp;`, nous pouvons appliquer le pragma `#pragma HLS RESOURCE variable=sum core=HAddSub_nodsp` à la variable sum pour contrôler l'opération d'addition en demi-précision.

Si les opérations de multiplication et d'addition étaient combinées en une seule instruction, il serait impossible d'appliquer des pragmas distincts à chacune des opérations. En utilisant le code de la figure 4, nous pouvons cibler spécifiquement chaque opération et forcer l'outil de synthèse à les implémenter sans utiliser de DSP, conformément aux objectifs de la solution 3.2.

Q3.3) Y a-t-il ici un avantage d'utiliser aucun dsp? Justifiez.

Oui, il y a des avantages à n'utiliser aucun DSP dans cette situation :

1. **Libération des ressources DSP** : Les DSP sont des ressources limitées sur un FPGA. En évitant leur utilisation pour les opérations arithmétiques en demi-précision, nous libérons ces blocs pour d'autres parties du design qui peuvent en avoir besoin, ou pour augmenter le parallélisme dans d'autres opérations critiques.
2. **Exploration architecturale** : Cela permet d'explorer les compromis entre l'utilisation des ressources logiques programmables et les DSP, et de trouver la configuration optimale en fonction des contraintes spécifiques du projet (comme la consommation d'énergie, la performance ou l'occupation des ressources).

Q3.4) Contrairement aux solution 1 et 2 (sections 1 et 2 resp. plus haut) est-ce que ici $II=2$ implique la moitié des ressources? Justifiez.

Non, dans le contexte de la solution 3.3, fixer l'Initiation Interval (II) à 2 n'implique pas nécessairement une réduction de moitié des ressources utilisées.

Justification :

- **Partage des ressources limité avec les LUTs et FFs** : Lorsque les opérations arithmétiques sont implémentées avec des ressources logiques programmables (LUTs et FFs), chaque opération nécessite son propre ensemble de ressources. Ces ressources ne sont pas facilement partagées entre différentes itérations du pipeline, contrairement aux DSP qui peuvent être partagés lorsque l'II augmente.
- **Impact de l'II sur les ressources** : Augmenter l'II (par exemple, passer de $II=1$ à $II=2$) signifie que le pipeline accepte une nouvelle donnée toutes les deux cycles d'horloge au lieu de chaque cycle. Cela peut réduire légèrement les ressources liées au contrôle du pipeline, mais les ressources principales pour les opérations arithmétiques restent nécessaires pour chaque opération, car elles ne peuvent pas être partagées efficacement.
- **Complexité des opérations en demi-précision** : Les opérations arithmétiques en demi-précision réalisées avec des LUTs et des FFs ont une complexité fixe en termes de ressources logiques. L'augmentation de l'II n'affecte pas cette complexité.

En conséquence, bien que l'augmentation de l'II puisse réduire légèrement certaines ressources, elle n'entraîne pas une réduction proportionnelle des ressources utilisées. Dans les solutions précédentes utilisant des DSP, augmenter l'II permettait de partager les DSP entre plusieurs opérations, réduisant ainsi le nombre total de DSP nécessaires. Ce mécanisme de partage est moins efficace avec les ressources logiques programmables.

Question 4

Q4.1) Expliquez le rôle des pragmas de ressources utilisées aux sections 4.2.1 et 4.3.1

Dans les sections 4.2.1 et 4.3.1, les pragmas de ressources sont utilisés pour contrôler et optimiser l'utilisation des ressources logiques (comme les LUTs et FFs) plutôt que des DSP. Cela permet de gérer la consommation des ressources FPGA, en optimisant le design pour réduire la dépendance sur les DSP, et en s'assurant que les calculs en 16 bits (type **short**) soient effectués via des opérations logiques dans les LUTs. Ces pragmas permettent également de réduire l'intervalle d'initialisation (II) pour un meilleur débit, tout en limitant l'empreinte matérielle sur le FPGA.

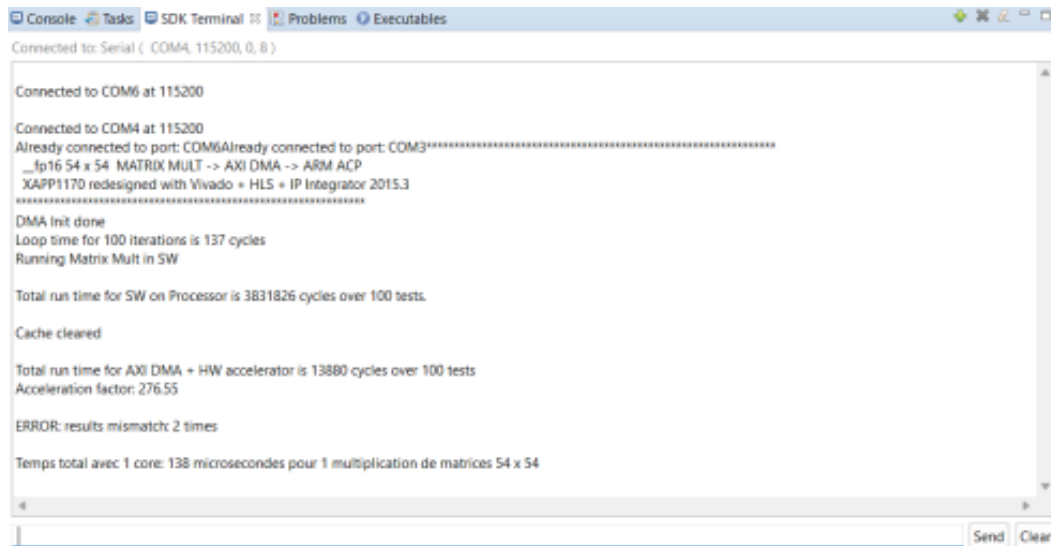
Q4.2) Y a-t-il ici un avantage d'utiliser aucun dsp? Justifiez.

Oui, il y a un avantage à n'utiliser aucun DSP dans cette configuration. Étant donné que les calculs en 16 bits (**short**) sont moins gourmands en ressources que ceux en 32 bits (**float**), les LUTs et les FFs du FPGA sont suffisants pour effectuer les opérations nécessaires. En évitant les DSP, on libère ces ressources pour d'autres tâches qui nécessitent des opérations de plus grande précision ou des calculs en virgule flottante, maximisant ainsi l'utilisation efficace des ressources du FPGA pour les opérations spécifiques.

Q4.3) Contrairement aux solution 1 et 2 (sections 1 et 2 resp. plus haut) est-ce que ici II=2 implique la moitié des ressources? Justifiez.

Non, ici II=2 ne signifie pas nécessairement la moitié des ressources. L'intervalle d'initialisation (II) de 2 cycles d'horloge indique seulement que chaque itération de boucle commence avec un décalage de 2 cycles, ce qui permet un certain niveau de parallélisme. Cependant, cela ne réduit pas proportionnellement le nombre de ressources utilisées, car la quantité de LUTs et FFs dépend également de la taille des données et de la façon dont les pragmas organisent les ressources. En d'autres termes, II=2 améliore le débit, mais les ressources consommées ne sont pas divisées par deux, car il y a des contraintes de parallélisme et de dépendances de données qui influencent l'utilisation globale des ressources logiques.

Q1S Concernant le projet_sol2_2 (Synthèse No1 Tableau 1):



```
Connected to Serial ( COM4, 115200, 0, 8 )

Connected to COM6 at 115200
Connected to COM4 at 115200
Already connected to port: COM6Already connected to port: COM3*****
fp16 54 x 54 MATRIX MULT -> AXI DMA -> ARM ACP
XAPP1170 redesigned with Vivado + HLS + IP Integrator 2015.3
*****
DMA Init done
Loop time for 100 iterations is 137 cycles
Running Matrix Mult in SW

Total run time for SW on Processor is 3831826 cycles over 100 tests.

Cache cleaned

Total run time for AXI DMA + HW accelerator is 13880 cycles over 100 tests
Acceleration factor: 276.55

ERROR: results mismatch 2 times

Temps total avec 1 core: 138 microsecondes pour 1 multiplication de matrices 54 x 54
```

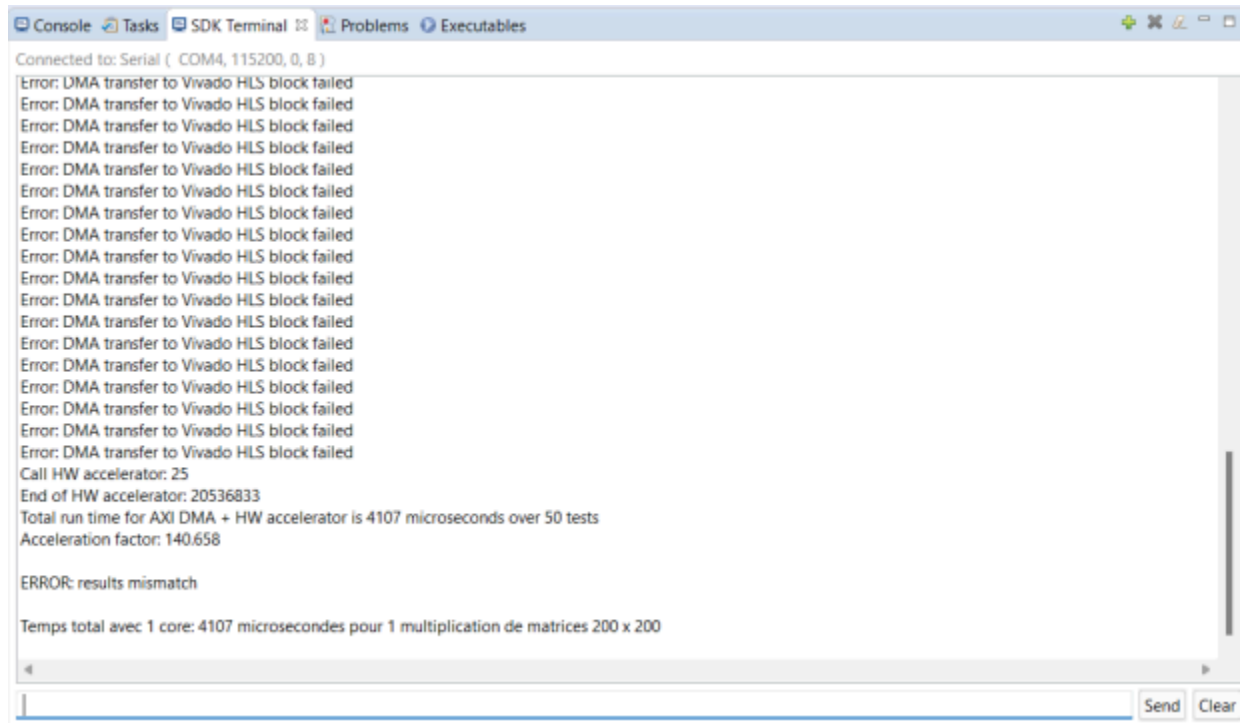
a) Que peut-on conclure des 2 erreurs de comparaison entre logiciel et matériel pour le projet_sol2_2?

Concernant les deux erreurs de comparaison entre le logiciel et le matériel observées dans le projet_sol2_2, cela peut être attribué à des différences dans la précision des calculs. Le matériel utilise des floats en demi-précision (FP16), ce qui peut entraîner des pertes d'exactitude par rapport aux flottants en simple précision (FP32) utilisés par le logiciel. Ces erreurs peuvent être accentuées dans des situations où des arrondis ou des valeurs limites interviennent dans les calculs. Aussi, il est possible que les matrices testées n'aient pas été correctement initialisées ou vidées, entraînant des incohérences entre les résultats du matériel et ceux du logiciel.

b) L'accélération du matériel par rapport au logiciel est-elle significative? Anticipez-vous une telle accélération à la suite de la section Exploration architecturale no 2 ? Justifiez.

L'accélération du matériel par rapport au logiciel est indéniablement significative. Le logiciel, exécuté sur le processeur ARM, a pris 3 831 826 cycles pour effectuer 100 tests, alors que le matériel, utilisant l'accélérateur hardware et l'interface AXI DMA, n'a pris que 13 880 cycles pour les mêmes tests. Cela correspond à un facteur d'accélération de 276,55 fois. Une telle accélération était anticipée, notamment en raison des optimisations effectuées lors de l'exploration architecturale no 2. L'utilisation d'une initiation d'intervalle (II) de 1 permet une parallélisation optimale des calculs, réduisant drastiquement la latence. De plus, l'intégration de l'interface AXI DMA a permis de minimiser les délais de transfert des données entre la mémoire et l'accélérateur matériel. En comparaison, le logiciel, étant séquentiel, est intrinsèquement moins performant pour traiter des matrices de cette taille. Ces résultats confirment l'efficacité des optimisations matérielles pour ce type de tâche.

Q2S Concernant le projet_sol3_3 (Synthèse No2 Tableau 1):



```
Connected to: Serial ( COM4, 115200, 0, 8 )
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Error: DMA transfer to Vivado HLS block failed
Call HW accelerator: 25
End of HW accelerator: 20536833
Total run time for AXI DMA + HW accelerator is 4107 microseconds over 50 tests
Acceleration factor: 140.658

ERROR: results mismatch

Temps total avec 1 core: 4107 microseconds pour 1 multiplication de matrices 200 x 200
```

a) Démarrez Vivado et ouvrez le projet `projet_sol3_3`. Dans le bas de la colonne de gauche, ouvrez l'onglet `IMPLEMENTATION` puis `Open Implemented Design` puis cliquez sur `Report Utilization`. Vous devriez voir apparaître la boîte de la figure 16 sur laquelle vous cliquez `OK`. Puis vous verrez apparaître le tableau de la figure 17 après avoir cliqué sur l'onglet `design_1_i`. En observant `HLS_accel_0` et `HLS_accel_1`, vous verrez qu'il y a un certain nombre de `BRAM`. Si vous ouvrez l'onglet `HLS_accel_0` y a-t-il une correspondance entre le nombre de `BRAM` et les `pragma HLS ARRAY_PARTITION` utilisés à l'Exploration architecturale no 3 (solution 3_3)? Justifiez.

Oui, il existe une correspondance entre le nombre de **BRAM** et les **pragma HLS ARRAY_PARTITION** utilisés dans l'Exploration architecturale n° 3 (solution 3.3). Les directives `set_directive_array_partition` ont divisé les dimensions des matrices `a` et `b` en blocs (block factor 14), ce qui augmente l'accès parallèle aux éléments de ces matrices. Cette augmentation des partitions se traduit directement par une utilisation accrue de la `BRAM`, car chaque partition nécessite une allocation séparée de mémoire. Ainsi, le nombre de blocs dans la directive correspond bien à la consommation observée dans Vivado pour le module `HLS_accel_0`.

b) Pourquoi ce nombre de BRAM observé dans Vivado n'était pas comptabilisé dans votre rapport de synthèse de Vivado_HLS (sous Summary)?

Le nombre de **BRAM** observé dans Vivado n'est pas comptabilisé dans le rapport de synthèse de Vivado HLS parce que le rapport HLS se limite généralement à évaluer les ressources nécessaires pour les éléments traités par le module de haut niveau, sans prendre en compte la structure exacte de l'implémentation générée dans Vivado. Les détails des interconnexions, les buffers supplémentaires et d'autres ajustements nécessaires pour la compatibilité FPGA sont ajoutés lors de l'implémentation dans Vivado, ce qui entraîne des différences.

c) En observant juste les ressources BRAM, aurait-on pu mettre 3 instances de hls_accel dans le projet projet_sol3_3? Justifiez.

En observant uniquement les ressources BRAM, il aurait été possible de mettre 3 instances de `hls_accel` dans le projet `projet_sol3_3`. Le tableau montre que chaque instance consomme 16 BRAM, et le SoC 7020 dispose de 140 BRAM. Ainsi, avec trois instances, la consommation totale serait de 48 BRAM, ce qui reste en deçà des ressources disponibles.

d) En observant juste les ressources de l'ensemble de la puce SoC 7020, aurait-on pu mettre 3 instances de hls_accel dans le projet projet_sol3_3? Justifiez.

En observant l'ensemble des ressources de la puce SoC 7020, il n'aurait pas été possible de mettre 3 instances de `hls_accel` dans le projet `projet_sol3_3`. La consommation de **Slice LUTs** est un facteur limitant. Chaque instance consomme environ 12 500 LUTs, et trois instances nécessitent environ 37 500 LUTs. Bien que cela soit techniquement possible avec un total de 53 200 LUTs disponibles, cela laisserait peu de ressources pour d'autres composants essentiels du design (comme les interconnexions, DMA, etc.), risquant ainsi de dépasser les limites pratiques pour l'intégration complète.

e) Sachant que j'ai une puce de la série Zynq-7000 SoC (chap. 1, p37) dans laquelle appartient la puce 7020 avec assez de ressources pour mettre plus de 4 instances sur de hls_accel. Qu'elle est alors ma limite en terme du bus AXI. Observez bien votre design (Fig 8). Justifiez.

La limite en termes de bus AXI est déterminée par la bande passante disponible et le nombre de maîtres et esclaves connectés au bus. Dans le design avec 2 instances de `hls_accel`, chaque instance est connectée à un DMA via une interface AXI. Si plus d'instances sont ajoutées, la bande passante du bus peut devenir un goulot d'étranglement, augmentant les temps d'accès mémoire et impactant ainsi les performances globales. Une augmentation du nombre d'instances nécessiterait de vérifier que la bande passante et les interconnexions restent suffisantes pour supporter toutes les transactions simultanées.

f) Finalement, est-ce que l'hypothèse de la page 16 est vérifiée? Justifiez.

L'hypothèse de la page 16 est vérifiée. En divisant les tâches entre deux cœurs, les calculs peuvent être exécutés en parallèle, réduisant ainsi le temps total requis par rapport à une exécution séquentielle sur un seul cœur avec $\Pi=1$. Cependant, l'efficacité réelle dépend également de la latence de synchronisation entre les cœurs et de la capacité du bus AXI à gérer les transferts simultanés.

Q3S Concernant le projet_sol4_2 (Synthèse No3 Tableau 1):

Si le DIM obtenu lors de l'exploration architecturale no 4 solution 4_2 est différent de celui proposé au Tableau 1 (DIM=200), expliquez pourquoi ce choix de 200 qu'on peut voir comme une valeur limite. Suggestion : allez consultez les ressources comme pour la question Q2S a).

Dans le projet_sol4_2, le choix d'une valeur DIM=200, bien qu'elle soit différente de la dimension optimale obtenue lors de l'exploration architecturale, peut être justifié en fonction des contraintes et des ressources disponibles sur la puce Zynq-7000 SoC (xc7z020). En examinant les rapports d'utilisation des ressources et de latence, ainsi que les directives utilisées, voici l'analyse détaillée :

1. **Contrainte des ressources matérielles :** Avec DIM=200, l'utilisation des DSP atteint 211 sur 220 disponibles, ce qui représente 95% de la capacité totale. Cela indique que cette dimension est proche de la limite maximale possible tout en respectant les contraintes d'utilisation des ressources DSP imposées par le FPGA. Toute augmentation supplémentaire de DIM pourrait entraîner un dépassement des ressources DSP ou une utilisation excessive de BRAM ou LUT, ce qui rendrait la conception non implémentable.
2. **Effet des pragmas ARRAY_PARTITION :** Les directives `set_directive_array_partition -type block -factor 100` appliquées sur les matrices a et b permettent de diviser les matrices en blocs de taille gérable par le pipeline. Cela optimise l'utilisation des DSP et réduit les conflits d'accès mémoire. Avec DIM=200, le facteur de partition est équilibré pour maximiser l'efficacité tout en évitant un dépassement de ressources.
3. **Pénalités de latence :** Le rapport montre que la latence pour une itération complète est de 44,107 cycles avec un initiation interval (II) de 1. Cela démontre que la pipeline est pleinement utilisée, ce qui minimise le temps d'exécution sans sacrifier les performances matérielles.
4. **Balance entre BRAM et LUT :** L'utilisation des BRAM et LUT reste modérée à 137 sur 140 et 5021 sur 53200, respectivement. Cela indique que le design DIM=200 atteint un équilibre raisonnable entre les ressources mémoire (BRAM) et logiques (LUT), optimisant ainsi la performance globale.
5. **Compatibilité avec le bus AXI :** DIM=200 garantit que la taille des transferts de données via le bus AXI reste dans les limites supportées par le contrôleur DMA, évitant tout goulot d'étranglement ou problème de dépassement de buffer.

En résumé, DIM=200 est choisi comme valeur limite car elle maximise l'utilisation des ressources disponibles sur la puce FPGA tout en respectant les contraintes de latence, d'occupation mémoire, et de bande passante AXI. Cette dimension offre une bonne balance entre la performance (grâce à un II=1) et les limitations matérielles imposées par le design et la plateforme.