|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SoC | Partitions |  | |  | Ressources sur FPGA | | | | | | | | |
|  |  | Estima-  Tion (fps) | Sur  Board (fps) | Précis-sion  (%) | Estimation | | | | Utilisation réelle | | | | |
| LUT | FF | BRAM | DSP | LUT | | FF | BRAM | DSP |
| N/A | Validation  sol séquentiel | 3.28 | N/A | N/A | N/A | | | | | | | | |
| N/A | Validation  sol multi\_thread | 82.02 | N/A | N/A | N/A | | | | | | | | |
| Zynq | all\_software  sol séquentiel | 14.47 |  |  | N/A | | | | | | | | |
| all\_software  sol multi\_thread1 | 14.41 | 16.96 | 84.96% | N/A | | | | | | | | |
| filter\_hw  (pipeline) | 26.13 | 21.18 | 76.63% | 13.26% | 10.12% | 50.36% | 4.55% | 15.55% | | 10.90% | 32.50% | 3.18% |
| canny\_hw  (loop\_tripcount) | 25.21 |  |  | 52.88% | 25.84% | 118.21% | 18.18% | N/A | | N/A | N/A | N/A |
| filter\_hw  (sans optimisation) | 12.08 |  |  | 12.82% | 10.04% | 50.36% | 4.55% | 15.46% | | 10.91% | 32.50% | 3.18% |
| filter\_hw  (autre optimisation) | 25.89 | 21.02 | 76.83% | 24.82% | 19.81% | 50.36% | 56.82% | 20.25% | | 13.92% | 32.50% | 56.36% |
| 2x\_filter\_hw  (pipeline) | 24.75 | 21.20 | 83.25% | 18.25% | 12.55% | 52.50% | 9.09% | 20.84% | | 14.31% | 33.93% | 6.36% |
| ZUS | zus\_all\_software | 51.10 |  |  | N/A | | | | | | | | |
| zus\_filter\_hw\_filtering  (pipeline) | 53.93 |  |  | 2.86% | 1.97% | 13.62% | 0.58% | | N/A | | | |
| zus\_2filter\_hw | 75.90 |  |  | N/A | | | | | | | | |

Pour faire le calcul de précision, nous avons utilisé la formule suivante :

**Commentaire et synthèse des résultats :**

* Validation solution séquentielle :
  + La validation séquentielle nous donne un résultat de 3.28 images par secondes lors de la simulation sur Spacestudio. Ce résultat est très bas, mais c’est attendu puisque l’algorithme séquentiel n’est pas performant.
* Validation solution multi\_thread :
  + Cette architecture est celle qui nous a donné les résultats les plus élevés. En effet, nous avons obtenu une moyenne de 82.02 images par seconde lors de la simulation sur Spacestudio. Cette architecture est très rapide puisque tous les modules sont en matériels et sont exécutés comme du code systemC sous Spacestudio. Toutefois, il est impossible de faire l’implémentation de cette solution puisque les modules réception0 et display0 doivent être en logiciel. De plus, il n’y aurait pas assez de ressources sur les cartes FPGA pour faire l’implémentation des tous les modules.
* All software solution séquentielle :
  + Cette architecture nous a donné de bons résultats avec une moyenne de 14.47 images par seconde. Cette architecture séquentielle est plus performante que la validation puisque le traitement des images est fait en logiciel, ce qui veut dire que les latences ajoutées pour faire la simulation en matériel n’est pas pris en compte et que seul le temps d’exécution sur le noyau linux est considéré. Dans ce cas, on peut alors estimer qu’il est plus rapide de faire le traitement d’image en logiciel pour la solution séquentielle.
* All software solution multi\_thread:
  + Cette architecture a produit 14.41 images par seconde lors de la simulation sur Spacestudio et 16.96 images par secondes lors de l’implémentation sur la carte FPGA. Nous avons calculé une précision de 84.96% entre nos résultats. Les résultats de cette architecture est très proche de la solution séquentielle. Pour voir un gain en performance avec la solution parallélisée, il faut mettre des modules en matériel.
* Filter hardware (pipline) :
  + Cette architecture utilise les modules filter0, reception0 et display0 en matériel et le reste des modules sont en logiciel. Nous avons obtenu 26.13 images par secondes pour la simulation sur Spacestudio ainsi que 21.18 images par secondes pour l’implémentation sur la carte FPGA, ce qui donne une précision de 76.63%. On peut voir que ces résultats sont meilleurs que l’architecture all software, puisque le module filter est plus performant en matériel. Pour ce qui est de l’utilisation des ressources, l’estimation montre que l’utilisation de la BRAM est à 50.3%, mais le reste des ressources sont assez basses. Toutefois, l’utilisation réelle des ressources montre que l’utilisation de la BRAM se situ plus autour de 32.50%, ce qui montre que Vivado HLS fait très probablement des optimisations lors de l’implémentation. Le reste des ressources est similaire à l’estimation.
* Canny hardware (loop\_tricount) :
  + Cette architecture utilise le module Canny0 en matériel et le reste des modules (sauf reception0 et display0) en logiciel. Nous avons obtenu 25.21 images par seconde avec la simulation de Spacestudio. Nous n’avons pas pu mettre cette solution en matérielle, car cette architecture utilise plus de ressources que celles disponible sur la carte FPGA. Pour l’utilisation des ressources, il n’était pas possible de vérifier l’utilisation réelle puisque l’implémentation de fonctionne pas si on dépasse les ressources disponibles. Cette solution utilise 118.21% de la BRAM disponible, car Canny utilise la BRAM pour stocker tous les résultats possibles du calcul de *atan* à la place de faire le calcul. C’est pour cette raison que l’utilisation de la BRAM est aussi élevée et qu’il est impossible de faire l’implémentation de cette architecture.
* Filter hardware (sans optimisation) :
  + Pour cette solution, nous avons obtenus 12.08 images par seconde dans la simulation de Spacestudio. Cette solution n’utilise aucun pragmas, ce qui augmente grandement le temps de calcul du module filter. L’utilisation des ressources est pratiquement semblable à la solution avec le pragmas Pipeline excepté que l’utilisation des LUT est un peu moins élevée. Toutefois, la perte de performance ne justifie pas l’économie de ressource de cette solution.
* Filter hardware (autre optimisation) :
  + Dans cette solution, nous avons mis un seul filtre en matériel mais nous avons utilisé un pragmas « #pragma HLS PIPELINE II=1 » à la boucle B et le pragmas « #pragma HLS unroll » sur la boucle C. Ces deux pragmas réduisent presque de moitié la latence d’exécution du filtre (1190431ns pour la solution pipeline et 518481ns pour cette solution). Toutefois, malgré cette diminution de latence d’exécution, nous n’avons vu aucune différence de performance tant en simulation qu’en implémentation sur la carte FPGA. Il est très possible qu’un autre module dans le système cause un gros goulot d’étranglement et qu’il n’est pas possible d’accélérer plus le système en ne modifiant que le filtre. Pour ce qui est de l’utilisation des ressources (estimation), si l’on compare à la solution de pipeline, nous utilisons presque le double des LUT, le double des flips flop et plus de dix fois plus de DSP. Pour ce qui est de l’utilisation réelle, la différence est beaucoup moins grande entre les deux solutions : 5% de plus de LUT, 3% de plus de FF, autant de BRAM, mais vingt fois plus de DSP. En bref, c’est l’utilisation des DSP qui a grandement augmenté avec cette optimisation, probablement causé par le pragmas Unroll qui demande que plus de calculs en parallèle soient exécutés en même temps. Nous avons obtenu 25.89 images par seconde en simulation et 21.02 images par seconde lors de l’implémentation sur la carte, ce qui donne une précision de 76.83%.
* 2 filter hardware (pipeline) :
  + Pour cette architecture, nous avons mis deux instances du filtre en matériel. Nous avons obtenu 24.75 images par seconde en simulation et 21.20 images par seconde en implémentation sur la carte, ce qui donne une précision de 83.25%. On peut observer qu’il n’y a pas vraiment d’avantage de performance comparé aux solutions précédentes (pipline et notre propre optimisation), ce qui est probablement dû à un goulot d’étranglement (bottleneck) dans un autre module de notre système. Pour l’utilisation des ressources, l’utilisation d’un filtre supplémentaire n’augmente que de quelques pourcents comparés à l’utilisation d’un seul filtre. Cela prouve encore une fois que le goulot d’étranglement du programme se trouve dans un autre module du programme.
* ZUS all software :
  + Pour cette architecture, nous avons utilisé le module Zynq Ultrascale+ qui est beaucoup plus performante que la Zynq. C’est pourquoi, dans cette solution, nous avons obtenu 51.10 images par seconde, ce qui est 3.6 fois plus élevé que la solution all software de la Zynq, ce qui prouve de la zus est beaucoup plus performante.
* Zus filter hardware (pipeline) :
  + Pour cette solution, nous avons obtenu 53.93 images par seconde, ce qui est environ le double des performances de la solution filter hardware sur Zynq. Pour ce qui est de l’utilisation des ressources, On peut voir que l’utilisation de toutes les ressource à grandement diminué comparément aux solutions sur Zynq, puisque la ZUS possède beaucoup plus de ressources matérielles que la Zynq.
* Zus 2 filter hardware (pipeline) :
  + Pour cette architecture, nous avons obtenu 75.90 images par seconde, ce qui est le deuxième résultat le plus élevé après notre architecture validation multi\_thread. Il est possible d’observer que l’ajout d’un filtre supplémentaire augment considérablement les performances, ce qui n’était pas le cas sur Zynq. Cela est causé par le fait que le goulot d’étranglement se trouve dans l’un des modules sur le Zus (master, resizing, canny, hough\_transformation) et que le Zynq n’était pas assez performant pour saturer une seule instance matérielle de filtering. Toutefois, avec le Zus qui est beaucoup plus performant, l’ajout d’un filtre supplémentaire en matériel donne de meilleures performances, ce qui montre que l’architecture précédente était limitée par le filtre en matériel.

Finalement, les meilleures architectures que nous avons pu tester sont celle qui utilisaient un filtre en matériel est qui utilisait le pragmas pipeline ou loop unroll. Il aurait été intéressant de tester les solution Zus en implémentation, toutefois il faudrait posséder l’un de ces cartes pour le faire.