TÉCNICO LISBOA

Parte 2 - Gauss Jacobi
Co-Projecto Hw/Sw

João Ramiro - 81138 José Carlos Vieira - 90900

Prof. Horácio Neto

---

1 Introdução

O projeto escolhido, dentro dos propostos, foi a implementação de uma arquitetura de co-processamento cujo seu objetivo é encontrar iterativamente uma solução de um sistema (diagonal dominante) de equações lineares, utilizando o método Gauss-Jacobi. O sistema, tal como requerido, pode ter até 100 equações lineares, onde tanto os elementos da matriz , como dos vetores são *floating points*. O sistema, calcula n iterações do método, até que atinja a condição de paragem associada, que incide no valor da norma da diferença entre a iteração anterior e a atual. Caso se trate de uma matriz não diagonal dominante, onde o método não é aplicável (não converge), terá um número máximo de iterações como condição final.

Dando seguimento ao trabalho desenvolvido na primeira parte do projeto, nesta, o objetivo incide na implementação de *hardware accelerators* na FPGA (*Field Programmable Gate Array*) ZYBO Zynq-7000 *Development Board*. Estes permitirão uma redução significativa do tempo de computação (*speedup*) do algoritmo Gauss-Jacobi. Para além destes, em contraste com a primeira parte do projeto, foi implementado o IP de multiplicação de *floating points* que permitirá realizar o cálculo implícito a cada iteração.

2 Algoritmo

Na álgebra linear numérica, o método Gauss-Jacobi (ou método iterativo de Jacobi) é um algoritmo para determinar as soluções de um sistema diagonalmente dominante de equações lineares $Ax = b$. O processo é iterado até convergir, ou seja, até Ax ser próximo o suficiente de b. Este algoritmo é uma versão simplificada do *Jacobi eigenvalue algorithm* [1]. O método tem o nome de Carl Gustav Jacob Jacobi.

> "Técnicas iterativas são raramente utilizadas para solucionar sistemas lineares de pequenas dimensões, já que o tempo requerido para obter um mínimo de precisão ultrapassa o requerido pelas técnicas diretas como a eliminação gaussiana. Contudo, para sistemas grandes, com grande percentagem de entradas de zero (sistemas esparsos), essas técnicas aparecem como alternativas mais eficientes. Sistemas esparsos de grande porte frequentemente surgem na análise de circuitos, na solução numérica de problemas de valor de limite e equações diferenciais parciais."[2]

Como referido, é considerado o sistema de equações: $Ax = b$, onde:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Então A pode ser decomposto num componente diagonal D e o resto R:

$$A = D + R \quad Onde \quad D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}, \quad R = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}$$

O sistema de equações lineares pode ser reescrito como:

$$Dx = b - Rx \tag{1}$$

O método de Jacobi é um método iterativo que resolve o membro esquerdo da expressão em ordem a x ao usar o método resultante da iteração anterior no membro direito. Analiticamente, isto pode ser escrito como:

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \tag{2}$$

ou, equivalentemente:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) \quad , i = 1, 2, \ldots, n. \tag{3}$$

Este método apenas converge para uma solução se, e somente se, o raio espetral de $D^{-1}R$ for menor que 1, i.e.:

$$\rho(D^{-1}R) < 1 \tag{4}$$

Consequentemente, o método de Jacobi é convergente sempre que a matriz A for diagonal dominante.

3 Implementação

Para a implementação do método Gauss-Jacobi foi desenvolvida uma arquitetura de co-processamento em *hardware* e *software*. Nesta arquitetura, o cálculo de maior complexidade, isto é, o produto interno entre a matriz quadrada A e o vetor x é efetuado pela PL (*programmable logic*) e os restantes cálculos implícitos a cada iteração são calculados pelo PS (*processing system*).

3.1 *Hardware*

Sendo o produto interno entre A e o x realizado na PL, foi necessária a introdução de alguns IPs (*Intellectual Propertys*). Como se pode observar pela figura 1, foram introduzidos dois multiplicadores, my_axis_float_matp, por se tratarem de valores do tipo *floating point*, onde, para a sua comunicação com o PS, é necessária a introdução de um AXI Interconnect e, consequentemente, um Processor System Reset. Para além destes, foram introduzidas duas AXI Direct Memory Access como *hardware accelerators* e consequentemente dois AXI SmartConnect.

Sendo necessário fazer uso da PL, é necessário ativar algumas funcionalidades no PS. Para a comunicação com o AXI Interconnect foi ativado o porto AXI_M_GP0 (*general-purpose*). Para o Processor System Reset, foi ativado um dos *clock reset*, FCLK_RESET0_N. Para cada um dos AXI SmartConnect foi habilitado um porto HP (*high performance*). Por último, foi habilitado o FCLK_CLK0 com uma frequência de 100MHz que serve como frequência de relógio da PL, sendo que este se liga aos relógios de cada interface AXI habilitada.

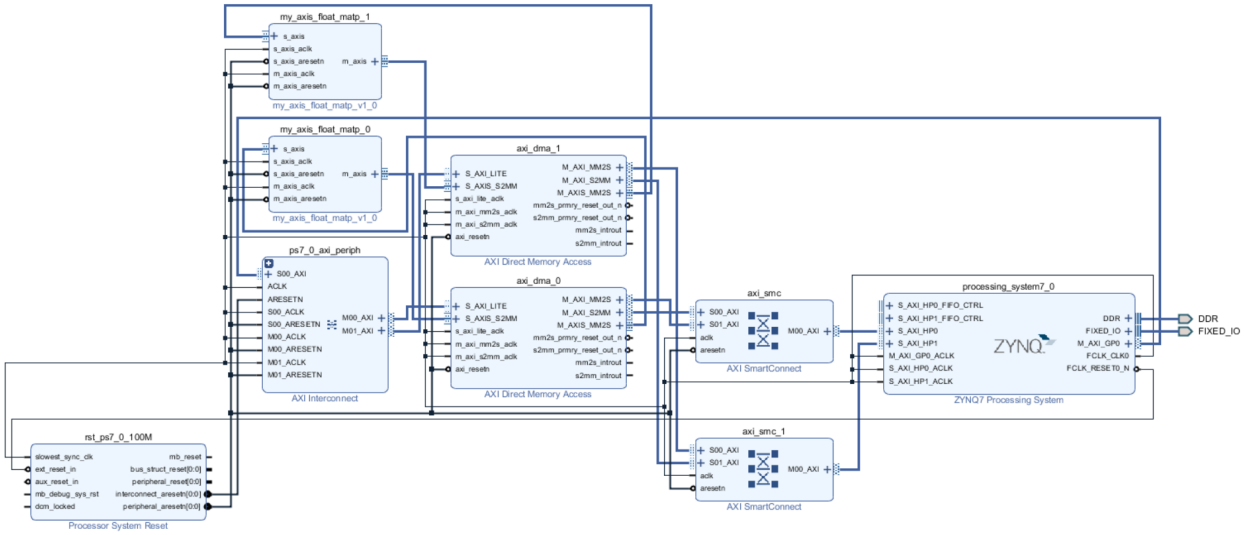

Figura 1: *Design* do *hardware* implementado.

3.1.1 Float IP

Comparativamente à primeira fase, foi trocado o Integer IP pelo Float IP, sendo que este é em tudo semelhante ao Integer IP. O vector x é enviado para a Memória B onde é armazenado durante a iteração. De seguida é enviada a matriz A linha a linha para o IP sendo que à medida que isto é feito, o produto interno dessa linha com o x armazenado na BRAM é calculado e posteriormente enviado por AXI stream de volta para a memória.

Este Float IP consiste portanto numa Memória, um multiplicador de floats, registos de pipeline e um acumulador.

Para o desenvolvimento do multiplicador, é necessário implementar o *datapath* que trata de toda a parte de registos, aritmética e operações lógicas, e a unidade de controlo (*Finite State Machine*) que faz uso dos *status* de saída do *datapath* para a escolha do estado atual e consequente controlo do *datapath*.

Datapath

O *datapath* do multiplicador implementado, é apresentado na figura 2. O seu objetivo é realizar o produto interno entre *data_A* e *data_B* e apresentar o resultado no *data_out*. Aplicando ao referido na secção 2, *data_A* será a_i e *data_B* será x_j.

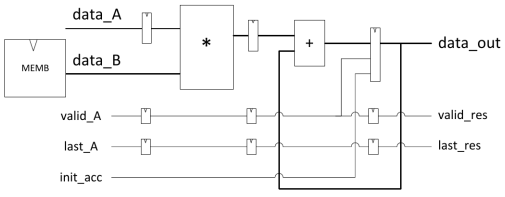

Figura 2: *Datapath* do multiplicador.

No produto interno é necessário realizar (em cada elemento) uma multiplicação e uma soma (acumulação). Para além destes, são também introduzidos os registos intermédios para reduzir o *critical path* e suportar *pipelining*. A sua utilização permite o aumento do *throughput*, isto é, resultados por unidade de tempo, ape-

sar da sua implementação causar um atraso de um ciclo de relógio por cada registo introduzido. Isto permite que a frequência se mantenha alta, permitindo por isso, uma aceleração do *hardware*, e consequentemente aumento do *throughput*.

No *path* dos sinais *valid_A* e *last_A* é necessário implementar tantos registos intermédios quanto os que existem no *path* dos dados. Como o *valid_A* serve para o controlo do registo de saída (a montante do somador), os *paths* têm de ser síncronos. Os registos intermédios dos *paths valid_A* e *last_A*, implementam um *shift-register* e concatenação do novo valor de entrada, a cada flanco ascendente do relógio. Ao mesmo tempo, o valor de saída do multiplicador e o *data_A* são escritos nas respetivas memórias a montante. O valor de *data_B* está sempre disponível, pois este está escrito na memória MEMB.

O primeiro passo para realizar o produto interno é fazer o *reset* à memória de saída do somador, pois, esta é utilizada na sua entrada e no princípio tem guardado um valor indefinido. Para isso é utilizado o sinal *init_acc* que é verificado a cada flanco ascendente do *clock*, que quando for 1, é feita a escrita de 0s.

Na saída do multiplicador estará sempre o resultado da multiplicação entre *data_A* e *data_B* e na saída do somador estará sempre a soma entre a memória de saída e a memória a montante do multiplicador, consequentemente é necessário determinar quando é efetuada a escrita do resultado na memória de saída. Para isso é utilizado o valor presente no segundo registo do *valid_A*. Em cada flanco ascendente do relógio é verificado o seu valor, e caso este seja 1, dá-se a escrita.

Control unit

De modo a que o multiplicador funcione corretamente, é necessário fazer uso de uma unidade de controlo que gere sinais de controlo e defina quando é que os dados devem ser enviados. Esta unidade corresponde à *Finite State Machine* descrita nas aulas teóricas. Esta máquina de estados tem portanto 3 estados: *st_read_A*, *st_read_B*, e *st_write* tal como se pode observar na figura 3.

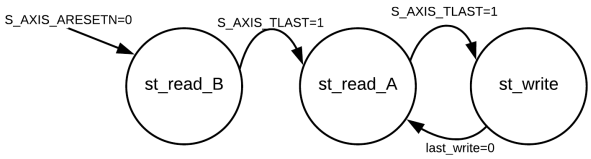

Figura 3: FSM do multiplicador.

A cada flanco ascendente do *S_AXIS_ACLK* é verificado o valor de *S_AXIS_TLAST*. Caso este tenha o nível lógico baixo, é colocada a máquina de estados no *st_read_B* onde, enquanto *S_AXIS_TVALID*=1 e *S_AXIS_TLAST*=0, todos os elementos recebidos (elementos de X), são escritos numa BRAM. Após a receção do sinal *S_AXIS_TLAST*=1, que indica que todos os elementos de X foram recebidos e escritos na memória, é passado para o estado st_read_A, onde são lidos os elementos de uma linha de A, sendo que a cada elemento lido, é feita multiplicação e a acumulação no *datapath*. Quando toda a linha é transmitida, passa-se ao estado de *st_write* onde o valor guardado no acumulador pode ser lido no *M_AXIS_TDATA*. Enquanto houver mais linhas de A, volta-se de novo ao estado *st_read_A* voltando se a fazer o produto interno dessa linha com X e posteriormente lido o resultado. O fim do produto interno das duas matrizes é dado pelo sinal *S_AXIS_TLAST*=1.

Test bench

De forma a verificar o correto funcionamento do Float IP foi utilizado o *testbench* fornecido. Neste *testbench* é feita a multiplicação de 2 matrizes 4 por 4 com elementos de *single precision*. O resultado pode ser observado nas figuras 4 e 5.

Pode ser observado, que inicialmente é enviado B (correspondente ao x_i) pois *selB* tem o valor lógico alto. De seguida é enviada cada linha de A separadamente, sendo que após a receção de cada elemento é feito o produto interno com B. No fim da receção de toda a coluna o resultado final é apresentado (*M_AXIS_TVALID* a 1).

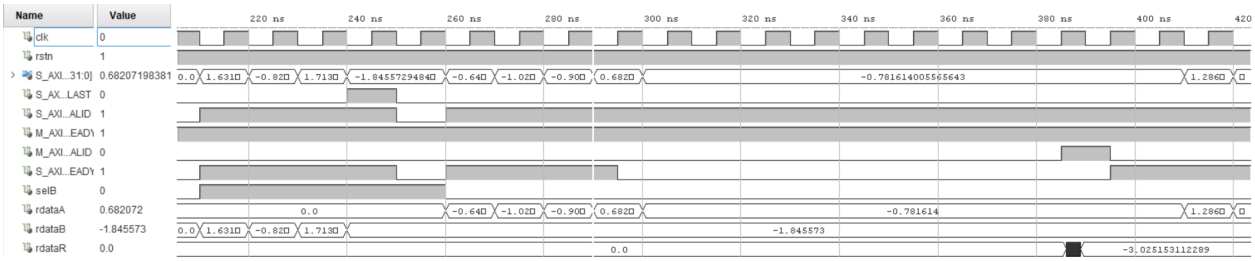

Figura 4: Cálculo de um valor.

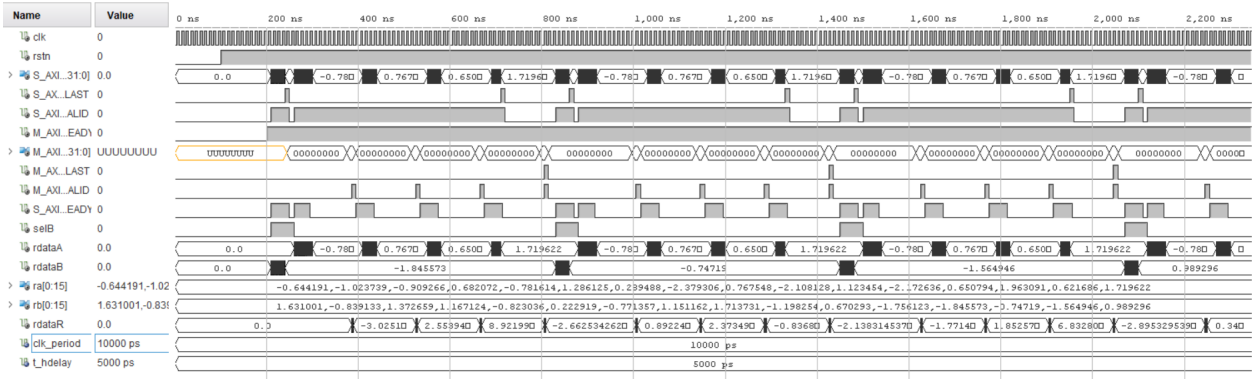

Figura 5: *Tesbench* Completo.

Este processo pode ser facilmente verificado na figura 4 onde, após a multiplicação entre $x_i = [1.63, -0.82, 1.71, -1.84]$ e a primeira linha de A $A(1,:) = [-0.64, -1.02, -0.90, 0.68]$, é obtido -3.025 e *M_AXIS_TVALID* a 1 aos 390ns. Para os cálculos dos próximos elementos, x_i não será enviado novamente, pois este está guardado em memória (BRAM).

3.1.2 DMA

Foi utilizada uma DMA para transferir os dados para o Float IP e posteriormente para a memória. Através da DMA é possível realizar as comunicações de forma muito mais rápida. Previamente os dados iam da memória para o processador por AXI-stream, e de seguida o processador enviava-os via AXI-Stream para o IP. Através da DMA, os dados passam diretamente para o IP, onde o processador apenas programa a DMA para definir o que precisa de fazer. O processador envia para esta por AXI4-Lite os comandos, ou seja, que informação deve ser transferida e para onde. A DMA envia os dados para o *hardware* via AXI-stream e para a memória passando primeiro pelo AXI Smart Connect.

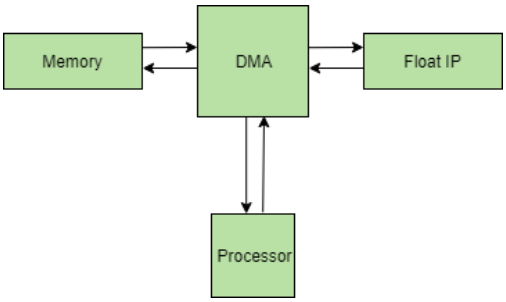

Figura 6: Esquema com DMA

3.1.3 2 Float IPs

No passo seguinte foi adicionado ao *hardware* outro Float IP. O esquema resultante encontra-se na figura 1. Este novo circuito inclui agora 2 Float IPs, 2 AXI smart connects e 2 DMAs responsáveis por conectar os *High Performance ports* da *board* por onde fluirão os dados para as DMAs que por sua vez transmitem os dados até aos Float IPs e que devolvem posteriormente os dados à placa. Finalmente existe um AXI Interconnect ligado à Zynq e a ambas as DMAs que servirá para as configurar.

3.2 *Software*

O primeiro passo é gerar e guardar em memória as matrizes necessárias para a execução do método. Tendo em conta o referido na secção 2, n pode tomar valores menores ou igual a 100 e a matriz A tem de ser quadrada e diagonal dominante. Para este passo as matrizes são guardadas num ficheiro binário que posteriormente é utilizado como *input* para inicializar a memória.

Foram utilizados os dois processadores disponíveis na ZYBO Zynq-7000 *Development Board*. Como era sabido em antemão que um dos processadores teria uma ocupação de programa muito maior que o outro, devido ao facto de apenas um deles ter acesso à porta série e consequentemente ser este a realizar os prints no terminal, foi decidido que este ocuparia a memória toda da ram_0 e o outro seria escrito no princípio da ddr_0.

Assim, foi utilizado como endereço base o 0x15000000 para o ficheiro input, que corresponde a uma posição de memória da *external memory* (DDR) suficientemente grande para guardar o programa do segundo processador e suficientemente distante do fim para guardar os dados *input*.

Para endereçar cada um destes vetores/matrizes, basta ter em conta o endereço anterior (ou o endereço base no primeiro caso) e o tamanho de cada um dos dados anteriores. Caso $n = 100$, os endereços de memória, são os apresentados no código 1.

```
#define MATSIZE     100
#define FLOAT_SIZE  4
#define MATRIX_SIZE FLOAT_SIZE*MATSIZE*MATSIZE
#define VECTOR_SIZE FLOAT_SIZE*MATSIZE

#define A_START_ADD   0x15000000
#define B_START_ADD   (A_START_ADD + MATRIX_SIZE)
#define X_START_ADD   (B_START_ADD + VECTOR_SIZE)
#define A_D_INV_ADD   (X_START_ADD + VECTOR_SIZE)
#define RES_START_ADD (A_D_INV_ADD + VECTOR_SIZE)
#define NORM_ADD      (RES_START_ADD + VECTOR_SIZE)

memA       = (float *)(A_START_ADD);
memB       = (float *)(B_START_ADD);
memX       = (float *)(X_START_ADD);
memRes     = (float *)(RES_START_ADD);
memA_D_Inv = (float *)(A_D_INV_ADD);
norm_value = (float *)(NORM_ADD);
```

Código 1: Posições de memória de cada um dos *inputs*.

O primeiro passo é invalidar toda a *cache* em ambos os processadores, que inclui tanto a L1 I-*cache* (instruções) e a L1 D-*cache* (dados) individual como a L2 *cache*, partilhada pelos dois processadores, que serve para dados e instruções.

Para além disto é definida uma zona de sincronismo, um semáforo, acessível por ambos os processadores. Para isso a MMU (*Memory Management Unit*) é configurada para desabilitar a *cache* na OCM (*On-Chip Memory*) nos endereços 0xFFFF0000-0xFFFFFFFF.

O objetivo, tendo dois processadores foi dividir tanto quanto possível, a parte de processamento por ambos, para que o paralelismo leva-se a uma redução do temporal no processamento requerido. De forma a reduzir as dependências entre os processadores, que implicitamente aumenta o número de *flushs* e/ou *invalidates* que têm um custo temporal associado, foi definido que cada processador controla a sua própria DMA. Assim, nesta fase inicial, cada processador

tem de realizar a inicialização da mesma. Esta inicialização inclui a procura da configuração no *hardware* para a instância do dispositivo, a configuração do *base address* do registo, a configuração dos dados da instância, a verificação de que o dispositivo esta em *simple mode* e o *disable* das interrupções pois é usado *polling mode*.

Após a inicialização, para dar início ao processamento do método Gauss-Jacobi, apenas falta um mecanismo de sincronismo entre os dois processadores, pois, o processador 0 como é o *master*, será quem carrega o programa para o processador 1, o que leva a um desfasamento entre os mesmos. Para isso é utilizada uma espera ativa, tendo como condição o valor presente no semáforo, em ambos os processadores até que estes estejam sincronizados.

Tendo em conta a equação 3, o somatório da multiplicação (produto interno) entre a_{ij} e $x_j^{(k)}$ é apenas aplicado quando $j \neq i$, ou seja quando não se trata de um elemento da diagonal de A. De forma a não fazer esta verificação por *hardware*, foram colocados a 0 todos os valores da diagonal de A e ao mesmo tempo, foi calculado o vetor correspondente ao inverso da diagonal de A via *software*. Este vetor é guardado de igual forma na memória DDR. Para isso, e de forma a dividir o processamento, cada processador executa metade desta operação e no fim faz o *flush* da sua metade da matriz A, para forçar a escrita dos dados em memória. No caso do vetor do inverso da diagonal não é feito o *flush*, pois cada um utilizará a sua metade, para o calculo da norma. Para o processador 0, o código é apresentado no código 2.

```
 for (i = 0; i < MATSIZE/2; i++){
        memA_D_Inv[i] = 1.0f/A(i,i);
         A(i,i) = 0;
 }
Xil_DCacheFlushRange((INTPTR)memA, (unsigned)(MATRIX_SIZE/2));
```

Código 2: Inverso da diagonal de A.

Após esta fase inicial, dá-se início ao processamento das iterações, que tem como objetivo a convergência dos valores Ax e B. Em cada iteração, é utilizado o mesmo mecanismo de sincronismo que o utilizado anteriormente, com o auxílio do semáforo, e posteriormente dá-se a programação da DMA. A programação inclui a submissão de uma transferência simples de todo o vetor x. Este vetor é enviado na sua totalidade pelos dois processadores para que seja feito o produto interno com as suas respetivas linhas da matriz A e de seguida é feita uma espera ativa até que o canal XAXIDMA_DMA_TO_DEVICE esteja disponível. Posteriormente é feito o *invalidate* do resultado do produto interno, memRes, para forçar a sua leitura da memória e de seguida a programação da sua receção, em que cada processador irá receber a sua respetiva metade. Por último é enviada, em cada processador, metade da matriz A e depois são feitas duas esperas ativas, uma em cada sentido, XAXIDMA_DMA_TO_DEVICE e XAXIDMA_DEVICE_TO_DMA, para ter a certeza que ambos os canais já terminaram de transmitir. No caso do processador 0, o código é apresentado no código 3.

```
*sync_f = PROC0_RESTARTED;
while (*sync_f != PROC1_RESTARTED);

status = XAxiDma_SimpleTransfer(&AxiDma_0,(UINTPTR)memX, VECTOR_SIZE, XAXIDMA_DMA_TO_DEVICE); // send memX
if (status != XST_SUCCESS) return XST_FAILURE;

while (XAxiDma_Busy(&AxiDma_0, XAXIDMA_DMA_TO_DEVICE)) { /* Wait for Tx*/ }

Xil_DCacheInvalidateRange((INTPTR)memRes, VECTOR_SIZE/2); // receive memRes

status = XAxiDma_SimpleTransfer(&AxiDma_0,(UINTPTR)memRes, VECTOR_SIZE/2, XAXIDMA_DEVICE_TO_DMA);
if (status != XST_SUCCESS) return XST_FAILURE;

status = XAxiDma_SimpleTransfer(&AxiDma_0,(UINTPTR)memA, MATRIX_SIZE/2, XAXIDMA_DMA_TO_DEVICE); // send memA
if (status != XST_SUCCESS) return XST_FAILURE;

while (XAxiDma_Busy(&AxiDma_0,XAXIDMA_DMA_TO_DEVICE)) { /* Wait Tx */ }
while (XAxiDma_Busy(&AxiDma_0,XAXIDMA_DEVICE_TO_DMA)) { /* Wait Rx*/ }
```

Código 3: Programação da DMA.

Depois de receber o resultado da multiplicação, dá-se início ao cálculo da norma da diferença entre a iteração anterior e a atual. Mais uma vez este trabalho é dividido pelos dois processadores, onde é feito o *flush* da sua metade valor de x (memX) calculado por *hardware*. No fim deste cálculo o processador 0, que é quem tomará a decisão de término ou continuação para a próxima iteração, para isso terá de receber o valor calculado pelo processador 1, onde, é utilizado outro mecanismo de sincronismo de espera ativa no processador 0. Assim, quando o processador 1 terminar o cálculo e atualizar tanto os valores de x como o valor da norma, o processador 0 é informado e aí tomará a decisão. Este processo, no caso do processador 0, encontra-se no código 4.

```
normVal = 0;
for (i = 0; i < MATSIZE/2; i++){
        x[i] = memX[i];
        memX[i] = (float)memA_D_Inv[i] * (memB[i]-memRes[i]);
        normVal += (x[i] - memX[i]) * (x[i] - memX[i]);
}

Xil_DCacheFlushRange((INTPTR)memX, (unsigned)(VECTOR_SIZE/2));
while (*sync_f != PROC1_COMPLETED);
Xil_DCacheInvalidateRange((INTPTR)norm_value, FLOAT_SIZE);

normVal += *norm_value;
n_it++;
```

Código 4: Cálculo da norma.

Este processo é repetido n iterações até que o valor da norma da diferença entre a iteração anterior e a atual seja suficientemente pequena ou até que seja atingido o número máximo de iterações. Quando terminar, o processador 0, através do semáforo, informa o processador 1 que terminou e este sai.

No fim verificou-se que o programa do processador 0 ocupa 0x1bf78 bytes e o do processador 1 ocupa 0xf460 bytes, ou seja poderiam ficar os dois na ram0.

4 Análise de Resultados

4.1 Verificação do *Hardware*

Na verificação do *hardware* é importante ter em conta dois grandes fatores, a análise temporal que nos permite verificar se é realmente possível implementar o desejado com a frequência de relógio desejada e ainda a ocupação da FPGA para verificar se é possível implementar o desejado ou ainda adicionar mais IPs de forma a conseguir uma melhor aceleração do *hardware*.

4.1.1 Análise Temporal

Após a interligação de todos os IPs, foram executados os *Run Synthesis*, *Run Implementation* e o *Generate Bitstream*. De forma a verificar que a frequência de relógio implementada, poderia efetivamente ser utilizada, foi verificado o *Timing Summary Report*, onde foram obtidos os valores apresentados na tabela 1. Como se tratam de tempos positivos, esta frequência pode ser utilizada. Caso contrário teria de ser diminuída, pois, o WNS corresponde à margem entre o *slowest delay path* entre registos e o fim do ciclo de relógio.

Grandezas	Tempos[ns]
WNS	0.698
WHS	0.023
WPWS	3.750

Tabela 1: *Timing Summary Report*.

4.1.2 Utilização da FPGA

Em relação à utilização da FPGA foram obtidos os resultados da tabela 2. Nesta pode ser verificado que foram utilizadas menos de metade das LUTs disponíveis, pelo que para obter uma maior aceleração, poderiam ser ainda adicionados mais dois Float IPs o que reduziria o tempo para quase metade. É importante notar que grande parte da utilização das LUTs e Registos resulta dos 2 AXI Smart Connects que ocupam 3852 LUTs e 3784 Registos. As DMAs também utilizam bastantes recursos sendo que as duas DMAs existentes utilizam 2813 LUTs e 3914 Registos. Em relação às BRAMs é usada uma Block RAM em cada Float IP para armazenar o vetor B sendo que as restantes são utilizadas pelas DMAs. Finalmente, são utilizados 4 DSPs em cada Float IP para efetuar a multiplicação de *floats*.

	Usados	Máximo
LUTs	8779	17600
Registos	9348	35200
BRAMs	6	60
DSPs	8	80

Tabela 2: Utilização FPGA.

4.2 *Speedup*

Após a implementação final, e de forma a verificar os *speedups*, foram comparados os tempos da implementação final com e sem *cache* e só de software. Na tabela 3 encontram-se se os tempos de cada implementação e o *speedup* relativo à implementação com software sem otimizações. Foi utilizada sempre a mesma matriz A de 100x100 e vetores B e x de 1x100. O algoritmo convergiu em 4 iterações.

Implementação	Tempo [us]	SpeedUp
Software Only (-O0)	3284	-
Software Only (-O3)	883.3	3.69
PS - PL s/ cache (-O0)	739.86	4.44
PS - PL c/ cache (-O0)	389.76	8.43
PS - PL c/ cache (-O3)	365.92	8.97

Tabela 3: *Speedups*.

Existem várias conclusões que se podem retirar desta tabela. Primeiro é que o uso da *flag* de otimização O3 é bastante mais útil na implementação apenas com software uma vez que causa um *speedup* de 3.69, enquanto que na implementação final o *speedup* é de apenas $8.97/8.43 = 1.07$. Além disso é importante o uso da *cache*, pois, para a versão com PS-PL (-O0) o uso da *cache* leva a um *speedup* $8.43/4.44 = 1.89$. Finalmente, comparando a versão PS-PL sem otimizações com *Software Only* sem otimizações, obtém-se um *speedup* de 8.43 o que é um resultado bastante positivo. Fazendo a mesma comparação mas compilando o software com a *flag* -O3 obtém-se um *speedup* de $8.97/3.69 = 2.43$ que apesar de não ser tão bom, continua a ser o resultado de sucesso.

Para analisar melhor as partes da implementação final que demoravam mais tempo, foram medidos os tempos de cada trecho do programa (tabela 4).

Trechos	O0 s/ cache [us]	O0 c/ cache [us]	O3 c/ cache [us]
Cálculos iniciais	113.178	89.898	89.225
Comunicações com as DMAs (1 iteração)	65.348	64.391	64.286
Calculos de estimativa de X (1 iteração)	87.006	9.708	4.135
Total	739.86	389.76	365.92

Tabela 4: Duração de cada trecho da implementação.

Relativamente aos cálculos iniciais, que incluem tanto a obtenção do inverso da diagonal de A como o sincronismo dos dois processadores, verifica-se que tem uma duração temporal significativa, pois, cada processador efetua 50 divisões

e de seguida efetuam uma espera ativa. É importante notar que estes cálculos geram um *overhead* inicial que afeta o *speedup*, especialmente quando o algoritmo efetua poucas iterações, tal como neste caso (4 iterações). No limite, quando o número de iterações tende para infinito, este *overhead* inicial é ignorado, e o *speedup* seria superior ao obtido.

Pode verificar-se também que em cada iteração, a parte que demora mais tempo são as comunicações com as DMAs, pois implicam não só o envio e receção dos dados da memória, como também a posterior atualização da *cache*.

Finalmente, visto que o cálculo da nova estimativa de x envolve apenas, para cada um dos 50 elementos em cada processador, uma subtração e de seguida uma multiplicação, este é relativamente rápido. Com a utilização da *cache*, existe uma diminuição significativa deste tempo e com a utilização da *flag* de otimização que permite o *reordering* das instruções e a *loop unrolling*, este cálculo demora menos de metade do tempo do que no caso sem otimizações.

Apesar de não terem sido obtidos estes tempos para a implementação com apenas 1 Float IP e 1 processador, esta implementação teve um *speedup* de quase 2 relativamente a essa versão, tal como era de esperar.

Além disso foi verificado que quase não houve diferenças de tempo entre a implementação de 1 Processador e 2 Float IPs e a implementação com 2 Processadores e 2 Float IPs. Isto deve-se maioritariamente ao custo temporal associado aos *flushes* e *invalidates* da *cache* que são necessários devido às dependências entre os dois processadores, que acabam por não compensar a divisão do processamento necessário.

5 Conclusão

Neste projeto foi desenvolvida uma arquitetura de co-processamento de HW e SW, com o objetivo de acelerar o método de Gauss-Jacobi. No *hardware* foram implementados 2 Float IPs e duas DMAs como *hardware accelerators* para realizar a multiplicação Ax, enquanto que por *software*, foram utilizados dois processadores, com mecanismos de sincronismo para realizar a norma da diferença entre a iteração anterior e a atual para decidir se o método já terá convergido ou não e assim verificar a condição de paragem. No fim foram obtidos resultados positivos, tendo-se verificado um *speedup* de 8.4 relativamente à versão de *Software Only* para as mesmas condições.

Gerando outro ficheiro input, onde o número de iterações para o cálculo do método fosse maior, iria minimizar o *overhead* inicial do programa, onde seria possível verificar uma melhoria no *speedup*.

Caso o tamanho da matriz fosse aumentado, não ultrapassando a capacidade da BRAM que guarda o vetor x, seria de prever, de igual forma, um aumento do *speedup*.

Foi ainda verificado que a utilização de um processador que controla as duas DMAs ou dois processadores, onde o processamento por *software* é dividido, tanto quanto possível, e onde cada um controla a sua própria DMA, não têm uma diferença significativa em termos de valores temporais. Isto deve-se principalmente ao custo temporal associado aos *flushes* e *invalidates*, necessários devido às dependências entre os dois processadores.

Num trabalho futuro, tal como verificado, seria possível implementar mais alguns *hardware accelerators* tal como a junção de mais dois Float IPs para o cálculo do produto interno, de forma a obter mais aceleração.

References

[1] `https://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm`

[2] `https://pt.wikipedia.org/wiki/Metodo_de_Jacobi`