Anderson Coimbra Mendes

Projeto de um Sistema Embarcado de Alto Desempenho em SoC para cálculo do UKF usando Co-processamento dedicado em hardware reconfigurável.

Belo Horizonte

2018

Anderson Coimbra Mendes

Projeto de um Sistema Embarcado de Alto Desempenho em SoC para cálculo do UKF usando Co-processamento dedicado em hardware reconfigurável.

Monografia apresentada durante o Seminário dos Trabalhos de Conclusão do Curso de Graduação em Engenharia Elétrica da UFMG, como parte dos requisitos necessários à obtenção do título de Engenheiro Eletricista.

Universidade Federal de Minas Gerais – UFMG

Escola de Engenharia

Curso de Graduação em Engenharia Elétrica

Orientador: Prof. Janier Arias García

Belo Horizonte

2018

Dedico esse trabalho ao meu irmão Arthur,
pela inspiração na profissão de engenheiro eletricista.

Agradecimentos

Agradeço ao Professor Janier García pela disponibilidade e orientação nesse trabalho. Agradeço aos meus mestres do curso de engenharia elétrica que indiretamente contribuíram para esse trabalho. E a minha família pelo constante apoio nessa jornada.

"You can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future."
(Steve Jobs)

Resumo

Os estimadores de estado têm um papel fundamental para sistemas embarcados principalmente para veículos aéreos não tripulados e carros autônomos, pois permitem determinar com precisão o estado do veículo como posição, velocidade e altura. O *Unscented Kalman Filter* (UKF) é um dos estimadores mais promissores, pois possui um bom desempenho comparado aos seus concorrentes, mas é notório seu custo computacional elevado e para sistemas de tempo real isso é crítico. O trabalho propõe uma solução para esse problema. O algoritmo foi analisado em relação ao tempo de processamento de suas funções, dessa forma foi determinado acelerar a decomposição de Cholesky. Dentre as metodologias possíveis para implementar um acelerador de hardware, foi escolhido utilizar um FPSoC com o chip Cyclone-V-SoC que integra uma FPGA e um microcontrolador(HPS) no mesmo chip. A arquitetura foi analisada e avaliou-se várias metodologias para transferir os dados entre a FPGA e a HPS. Um co-processador dedicado para calcular a decomposição de Cholesky foi projetado e o circuito digital foi testado na prática utilizando o kit de desenvolvimento DE0-Nano-SoC. Pode-se observar uma aceleração de 2006% para o Cholesky, o que representou uma aceleração final de 26% para o algoritmo do UKF.

Palavras-chaves: UKF. Cholesky. Acelerador de Hardware. Cyclone V SoC. DE0-Nano-SoC. FPSoC. Sistemas Digitais. Sistemas Embarcados. VANT.

Abstract

States estimators play a key role in embedded systems primarily for autonomous unmanned aerial vehicles and cars, as they allow the determination of vehicle state such as position, speed, and height with a higher accuracy. The Unscented Kalman Filter (UKF) is one of the most promising estimators as it performs better compared to its competitors, but his high computational cost is notorious and for real-time systems this can be critical. This paper proposes a solution for this problem. The UKF algorithm processing time was analyzed, and the Cholesky decomposition was chosen to be accelerate as it consumes more operational time. Among the possible methodologies to implement a hardware accelerator, it was chosen to use an FPSoC with the Cyclone-V-SoC chip that integrates an FPGA and a microcontroller (HPS) on the same chip. The micro-architecture was analyzed and several methodologies to transfer the data between the FPGA and the HPS were evaluated. A dedicated co-processor for Cholesky's decomposition calculation was designed and the digital circuit was tested using the DE0-Nano-SoC development kit. We could see an acceleration of 2006 % for Cholesky, which represented a final acceleration of 26 % for the UKF algorithm.

Key-words:UKF. Cholesky. Hardware Accelerator. Cyclone V SoC. DE0-Nano-SoC. FPSoC. Digital Systems. Embedded Systems. VANT.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Sistemas embarcados são sistemas de computadores dedicados a executar uma função específica. Seu surgimento se deu devido ao avanço da eletrônica e computação, possibilitando a criação de dispositivos inteligentes e sistemas eletrônicos capazes de executar tarefas em tempo-real. Eles controlam muitos dispositivos hoje em dia, focando em eficiência energética, pequenos tamanhos e capacidade de processamento significativa.

Um sistema embarcado é, em sua maioria, baseado em microcontroladores que são compostos de um ou mais microprocessadores e periféricos no mesmo circuito integrado. Dependendo do nível de processamento requerido pela aplicação, alguns sistemas embarcados possuem processadores mais potentes e seus periféricos são circuitos integrados separados. Diferentemente do mercado de computadores e notebooks, diferentes arquiteturas são utilizadas, uma vez que o sistema embarcado é voltado para uma aplicação específica e não é um produto genérico.

Um dos primeiros sistemas embarcados modernos reconhecidos foi o Apollo Guidance Computer que foi desenvolvido por Charles Stark Draper em 1965. Os desafios iniciais para sua criação eram enormes na época sendo considerado a parte do projeto da Apollo mais complexa e com chance de falha, pois era uma inovação a utilização de circuitos integrados monolíticos (WIKIPEDIA, 06/2018).O primeiro microprocessador, o Intel 4004 foi projetado para ser utilizado em calculadoras e pequenos sistemas. Em 1978, a National Engineering Manufactures Association desenvolveu um modelo padrão para microcontroladores programáveis que incluía diversas aplicações.

Essas aplicações iniciais eram muito custosas, mas houve uma queda muito grande nos preços e um aumento no processamento. Hoje em dia a facilidade e a quantidade de microcontroladres é muito grande, logo a indústria eletrônica tem sido fortemente impulsionada pela grande demanda de sistemas embarcados sendo produzidos em massa. Eles são desde dispositivos portáteis como MP3 players, computador de bordo, até sistemas maiores e mais complexos como controladores de sistemas aeronáuticos e industriais.

Existem classificações dentro de sistemas embarcados, uma delas são os sistemas embarcados de tempo real que exigem uma tomada de decisão dentro de tempo máximo pré-determinado. Pode-se categorizar um sistema de tempo real em sistemas críticos ou não-críticos, em que no sistema crítico caso um desvio desse tempo pode gerar um agravamento sério como instabilidade do sistema ou até mesmo acidentes que põem em risco a vida de pessoas (BURNS, 2009). Existem vários sistemas críticos presentes no dia-a-dia das pessoas como por exemplo os sistemas automobilísticos: o controle de tração, estabilidade e ABS exigem uma atuação muito rápida evitando a perda de tração, estabilidade

ou que as rodas do carro travem. Se esses sistemas não atuarem dentro do tempo previsto é possível que acarrete em um acidente devido à perda da tração, estabilidade e maior tempo de frenagem no caso do ABS (BURNS, 2009).

Carros autônomos, VANTs (veículo aéreo não tripulado), foguetes militares e satélites também possuem diversos sistemas críticos de tempo real. Grande parte deles exigem que o estado atual como posição/velocidade/aceleração seja preciso para que possa tomar decisões como mudança de trajetória. Os sensores (GPS, acelerômetro, etc.) por sua vez apresentam um erro considerável para esse tipo de sistema e para solucionar isso é possível utilizar algoritmos para estimar o estado atual e corrigir a imprecisão. Os estimadores de estado, utilizam as medições das grandezas físicas atuais e dados anteriores para então calcular com maior precisão o estado atual. Em um exemplo prático, os estimadores de estado determinam a posição mais precisa de um VANT que utiliza esse dado no cálculo da sua trajetória e verifica se precisa tomar uma decisão como desviar de um obstáculo. Sem o estimador, o erro de posição pode impedir que o VANT desvie de um objeto e colida.

Por essa necessidade de precisão em sistemas críticos, a demanda por estimadores de estado rápidos e precisos tem aumentado. Um método de estimação bem popular para muitas aplicações tem sido a família de algoritmos do filtro de Kalman, particularmente o filtro de Kalman estendido (EKF) e, mais recentemente, o filtro de Kalman Unscented (UKF) tem ganhado notoriedade(GUSTAFSSON, 2012).

As aplicações de sistemas embarcados que utilizam estimadores de estado exigem um alto desempenho computacional para execução dos algoritmos no tempo previsto, mas ao mesmo tempo necessitam ter outras importantes características: ser fisicamente pequenos e leves para não inviabilizar o produto, por exemplo: placas eletrônicas grandes e pesadas em drones causam um desafio no projeto aerodinâmico; o custo deve ser pequeno pois um alto custo de produção também inviabilizaria o produto; o consumo de energia deve ser o menor possível pois em sistemas embarcados a energia disponível é limitada, geralmente o suprimento de energia é via bateria que possui uma capacidade energética muito pequena pois que baterias com grande potencial energético são custosas, volumosas e pesadas.

Esse trabalho está inserido nesse contexto de sistemas críticos de tempo real e confronta o problema do custo computacional do UKF para sistemas embarcados menores propondo um processamento eficiente do algoritmo. O tema escolhido foi o "Projeto de um Sistema Embarcado de Alto Desempenho em SoC para cálculo do UKF usando Co-processamento dedicado em hardware reconfigurável".

Então esse trabalho tem como objetivo principal acelerar o algoritmo de UKF para que possa ser viável sua implementação em pequenos sistemas embarcados como drones. Como obejetivos secundários o trabalho visa entender as características do algoritmo

UKF traçando seu perfil para identificar o gargalo no custo computacional, projetar um co-processador que possa executar cálculos em paralelo para garantir a performance e implementar todo sistema em um kit de desenvolvimento.

Além de propor essa solução e ainda sobre objetivos secundários, o trabalho visa a melhor compreensão de como utilizar o chip Cyclone-V-SoC, definindo uma metodologia para a criação de um projeto no chip Cyclone-V-SoC, bem como métodos de transferência de dados entre o processador principal e o co-processador com um estudo do melhor fluxo para uma dada aplicação. Isso permite que novos projetos de sistemas embarcados com aceleradores possam ser feitos com mais agilidade.

O trabalho está estruturado da seguinte forma: O capítulo 2 é uma introdução teórica necessária para o entendimento do trabalho, bem como apresenta as mais atuais técnicas e tecnologias utilizadas em sistemas embarcados e processadores

O capítulo 3 apresenta todo o desenvolvimento do projeto passando pela análise do custo computacional do UKF, análise do chip, escolha do fluxo de transferência de dados, arquitetura do co-processador, definições de projeto e implementação prática.

O capítulo 4 apresentará os resultados e o capítulo 5 a conclusão desse trabalho.

2 Embasamento Teórico

Em primeiro lugar é preciso compreender os conceitos básicos a seguir para o mínimo de entendimento desse trabalho.

- Processador

  Processador é o componente de hardware que processa os dados. Ele executa instruções que são determinadas via software. Essas instruções que são em sua essência cálculos matemáticos como somar, subtrair, multiplicar, dividir, comparar, entre outros. Além disso possui outros tipos de instruções como gravar dado na memória, pegar dado da memória e pular para outra instrução. Com esses elementos básicos o processador consegue executar uma enorme quantidade de dados. A quantidade de instruções depende da arquitetura de cada processador, alguns possuem mais outros menos.

  Características importantes:

  - Frequência (Clock): A velocidade de processamento do processador é proporcional a frequência de clock que ele trabalha.
  - Cache: Memória interna do processador, utilizada para armazenar dados temporários de processamento. A cache possui um alto desempenho, sendo bem mais rápida que as outras memórias. Fica localizada fisicamente próxima ao núcleo do processador para evitar que quando a CPU necessitar de dados, ela não demore para pega-los.
  - ULA: A unidade lógica aritmética é o circuito responsável pelas operações matemáticas da CPU. A instrução determina qual o tipo de operação e em qual endereço de memória os dados estão. Assim a ULA executa a operação com os dados de entrada.

- Co-Processador / Acelerador de hardware

  Co-processadores são basicamente um processador dedicado para resolver algum cálculo não trivial, que para um processador genérico possui inúmeras operações matemáticas. Ou seja, do ponto de vista do processador principal, eles podem ser interpretados como uma extensão da ULA. Os co-processadores são aceleradores de hardware utilizados para criptografia por exemplo, pois os cálculos são demorados se executados em um processador comum.

- FPGA / Hardware Reconfigurável

Uma FPGA (Field Programmable Gate Array) é um circuito capaz de implementar circuitos lógicos. Consiste em um arranjo de células lógicas capazes de se organizar de acordo com o desejado. Uma FPGA pode ser reconfigurada quantas vezes o usuário desejar, assim é possível implementar circuitos digitais de uma forma prática e rápida.

- SoC

  Um SoC (system-on-a-chip) é um sistema inteiro dentro de um único circuito integrado. Ele possui diferentes blocos digitais e analógicos. Sistemas embarcados geralmente possui um SoC, como por exemplo um microcontrolador mais um *transceiver* de radiofrequência para comunicação no mesmo chip.

- Memórias

  Existem vários tipos de memórias em um sistema, cada uma com sua especificação. Normalmente, quanto maior sua capacidade em bits, mais lenta a memória é. Elas podem ser voláteis ou não. As memórias flash são não voláteis e armazenam os dados que não podem ser apagados. Também armazena as instruções a serem executadas pelo processador.
  As memórias RAM e a cache são voláteis e armazenam dados temporários de processamento.

- DMA

  A DMA é um componente que permite o acesso direto dos periféricos as memórias. Normalmente, o acesso as memórias é feito pela CPU que pega ou grava dados. Mas se um periférico quiser acessar a memória ele deve esperar a CPU para que ela faça isso, seja gravar ou pegar um dado em uma posição de memória. A DMA permite que o periférico como por exemplo um sensor de temperatura grave seu dado na memória sem precisar esperar a CPU, enquanto isso a CPU pode executar uma outra atividade melhorando a performance geral ou até a CPU pode ficar em modo *sleep*, economizando energia.

Em uma solução para um sistema embarcado, um sistema é proposto analisando vários parâmetros como poder de processamento, custo, tamanho, consumo, requisitos de memória, protocolos de comunicação, sensores e conversores. Em sistemas em que exige uma necessidade de resposta rápida como é o caso de sistemas críticos em tempo-real existem alternativas para resolver essa limitação da melhor forma possível. Alguma das soluções utilizadas atualmente para um melhor processamento são:

1. Aumento do clock da CPU
   Um aumento no clock da CPU permite que mais instruções sejam realizadas por

segundo, ou seja, dobrando o clock se dobra o poder de processamento. Essa abordagem tem grandes desvantagens: maior tamanho físico, maior área de silício, maior custo, aumento da complexidade do sistema pois quanto maior a frequência mais complexo serão os osciladores e a pll para gerar esse clock alto, e maior o consumo. Além disso existe uma limitação física em que se pode aumentar o clock sem perturbar o sistema.

2. Multi-cores
   Uma abordagem recente que se tornou comum nos computadores pessoais é a utilização de mais de um core e o processamento é feito em paralelo. A desvantagem é que a complexidade do sistema aumenta muito, pois nem todas as sequencias de instruções podem ser quebradas e paralelizadas. Também ainda existe o compartilhamento de recursos como memória. Outra desvantagem é um grande aumento no custo e tamanho físico.

3. Processadores especializados
   Processadores com funções definidas estão cada vez mais sendo utilizados pois eles são um bom balanceamento de tamanho físico, custo, performance e consumo. Por exemplo: GPUs (graphics processing unit) dedicado a processamento de imagens ou aceleradores de hardware e coprocessadores especializados em resolver alguma tarefa. Em sistemas embarcados com criptografia a maioria utiliza co-processadores pois além de ganhar em performance o custo é menor do que resolver em um processador genérico.

Hoje em dia fabricantes oferecem soluções em chips únicos combinando processadores de uso geral, processadores especializados e periféricos resultando no que é conhecido como system-on-chip (SoC). Os SoCs permitem que o consumo de energia e o tamanho da placa sejam reduzidos, além da confiabilidade e desempenho serem superiores em comparação com uma solução multichip equivalente (MOLANES, 2018).

Dependendo da aplicação, algumas abordagens são mais adequadas que outros. Por exemplo, as GPUs geralmente desempenham melhor que FPGAs para cálculos de ponto flutuante, mas FPGAs podem ser usados com vantagem em aplicações onde a arquitetura fixa da GPU não se ajusta(ASANO; YAMAGUCH, 2009). Além disso, as FPGAs são reconfiguráveis e quase sempre mais eficiente em relação ao consumo de energia, operações por segundo, operações por watt do que as GPUs (KESTUR; WILLIAMS, 2010). Uma desvantagem processadores especializados é a necessidade de software personalizado para extrair o máximo da arquitetura, complicando o desenvolvimento e a portabilidade de aplicativos. Existem linguagens padrão e eficientes (como CUDA ou OpenCL) que podem ser usadas com GPUs, bem como compiladores que aproveitam automaticamente os coprocessadores que são acessados via instruções dedicadas. Em contraste, até recentemente

as FPGAs eram principalmente programado em linguagens de descrição de hardware (HDLs), o que implica um maior tempo de colocação no mercado e exige dos projetistas conhecimento sobre detalhes de hardware para que o desempenho possa ser otimizado, resultando que em FPGAs sendo menos usados do que outras soluções (RODRıGUEZ-ANDINA; MOURE, 2015).

Para otimizar o processamento de dados em processadores genéricos e especializados as técnicas abaixo são comumente aplicadas.

- Pipeline

  Pipeline é uma técnica utilizada em processadores para aumentar a performance. Consistem basicamente em quebrar as etapas de processamento para que em cada etapa exista uma instrução diferente em andamento. Em um processador sem pipeline é necessário que uma instrução seja totalmente executada antes que uma nova possa ser processada. Com o pipeline existe uma quebra nas etapas, assim em cada etapa existe uma instrução em andamento e não existe a necessidade de esperar uma instrução passar por todas as etapas antes de uma outra começar.

- Processamento em paralelo

  Dois ou mais dados são processados no mesmo instante de tempo. Por exemplo, um processador executa duas instruções ao mesmo tempo.

Em relação aos algoritmos estimadores de estado o Kalman Filter foi uns dos mais utilizados, mas funciona muito bem apenas para funções lineares. A maior parte dos problemas do mundo real são não lineares, o que torna o Kalman Filter ineficaz. Pensando nisso uma melhoria foi feita e o Extended Kalman Filter(EKF) foi criado. O EKF utiliza a Série Taylor (e da Matriz Jacobiana) para aproximar linearmente uma função não linear em torno da média do Gaussiano e depois predizer os valores, se tornando uma técnica muito utilizada (GUSTAFSSON, 2012).

O algoritmo Unscented Kalman Filter (UKF) também é da família de Kalman e foi proposto por Julier and Uhlman. É uma versão melhorada do EKF que visa resolver seu problema de linearização. O EKF utiliza apenas um ponto (media da gaussiana) para a linearização o que em sistemas altamente não lineares apresenta um desempenho ruim (JULIER, 1997).

O UKF usa uma técnica de amostragem determinística conhecida como unscented transform (UT) para escolher um conjunto mínimo de pontos de amostra (chamados de sigma points) em torno da média. Os sigmas points são então propagados através das funções não lineares, a partir das quais uma nova média e estimativa de covariância são geradas (JULIER, 1997).

Para determinados sistemas, o filtro UKF estima com mais precisão a verdadeira média e covariância. Isto pode ser verificado com amostragem de Monte Carlo ou expansão da série posterior de Taylor. Além disso, essa técnica elimina o requisito de calcular explicitamente os jacobianos como no EKF, o que para funções complexas pode ser uma tarefa difícil (ou seja, exige derivações complicadas se feito analiticamente ou pode ser computacionalmente caro se feito numericamente), se não impossível (se essas funções forem não diferenciáveis)(GUSTAFSSON, 2012).

3 Metodologia

O trabalho consiste em três grandes partes: 1- Escolha da topologia e definições da arquitetura do sistema; 2- Projeto do co-processador; 3- Implementação no SoC.

3.1 Escolha da topologia e definições da arquitetura do sistema

A definição da topologia utilizada é uma parte importante dentro do trabalho, pois ela determina variáveis relevantes para o projeto do co-processador que serão discutidas a seguir. Além disso a performance, consumo, tamanho físico e custo estão atrelados a topologia utilizada, então além de uma análise técnica, é preciso pensar na viabilidade do produto.

Pensando no sistema como um todo, existem duas formas de implementar o co-processador: por *Hard-IP*, desenvolvimento de um co-processador CMOS fisicamente em silício; Ou por *Soft-IP*, implementação em um hardware reconfigurável. A performance em *Hard-IP* é bem superior ao de hardware reconfigurável, mas é inviável seu projeto visto que é necessário a fabricação do dispositivo em silício que é caro e possui mais etapas de projeto como layout. Dessa forma, para prototipagem é muito comum a utilização de hardware reconfigurável tanto academicamente quanto na indústria.

Então, a ideia é utilizar um hardware reconfigurável para facilitar a implementação do co-processador, além de permitir uma rápida depuração e viabilizar a execução. Visto isso, existem pelo menos três formas de implementação:

1. FPGA

   Implementação de todo o sistema em hardware reconfigurável, inclusive o processador principal. Essa proposta facilita a comunicação entre o co-processador, processador, memórias e periféricos. Pois ela é altamente configurável e o barramento de comunicação não é um problema. Sua desvantagem é que necessitaria de muitas células da FPGA, o chip é custoso e seu tamanho físico é bem considerável.

2. FPGA + Processador

   Implementação somente do co-processador em uma FPGA e comunicação com um microcontrolador através de um barramento, ver figura . O custo do sistema cai consideravelmente e o tamanho também, mas existe uma grande rigidez na comunicação dos blocos pois já são predefinidas e não se pode configurar. A comunicação entre o co-processador e o processador é ruim pois existe uma grande latência já que ela é externa.

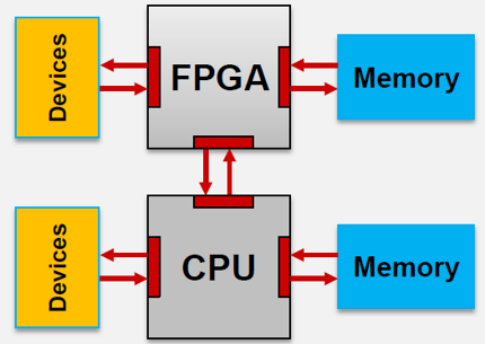

Figura 1 – Arquitetura: FPGA + Processador.(INTEL, 11/2018)

3. FPSoC

   Com o avanço de tecnologias e processos para fabricação de semicondutores uma nova categoria de SoC surgiu. Consiste em um único chip com processador e periféricos, mas também possui um bloco de hardware reconfigurável. O custo é intermediário, existe uma flexibilidade intermediária de comunicações e canais dedicados para suprir a deficiência na comunicação como um canal direto da FPGA para a cache do processador principal, canal direto para a memória RAM. A figura 2 mostra essa arquitetura.

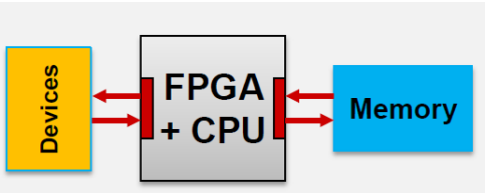

Figura 2 – Arquitetura: FPSoC.(INTEL, 11/2018)

Foi definido que a arquitetura FPSoC é a melhor opção e dentre os chips possíveis foi escolhido o Cyclone V SoC da Altera, com o kit DEO-Nano SoC. O motivo foi sua alta configurabilidade, menor latência na comunicação, alta performance, o custo é relativamente baixo e é uma tecnologia inovadora que possui poucos estudos publicados, portanto será extremamente proveitoso desenvolver nesse SoC.

Com a topologia escolhida, a seção 4.2 visa entender a arquitetura do chip e especificar os requisitos do sistema. É sabido que a comunicação entre o processador principal e o co-processador é a parte mais delicada de todo o sistema, é necessário um bom entendimento do chip escolhido, dos barramentos e de como funciona o fluxo de processamento. Foram analisadas a arquitetura de topo do chip, as funcionalidades e apresentado em 4.2.2, para determinar os seguintes parâmetros:

- O barramento que deve ser utilizado entre a FPGA e o processador principal.

- A largura de banda dos canais de comunicação.
- Quantidade de células disponíveis na FPGA.
- Frequência dos clocks do barramento, core e FPGA disponíveis.
- Melhor fluxo de processamento e transferência de dados.
- Determinar quantas palavras podem ser escritas e lidas na FPGA por segundo.

3.2 Projeto do co-processador

O projeto do co-processador pode ser dividido em três partes:

- Análise do algoritmo a ser acelerado e decomposição em blocos matemáticos.
- Criação da micro-arquitetura do co-processador.
- Criação dos blocos em Verilog.

O algoritmo que foi determinado anteriormente para ser acelerado foi decomposto em blocos matemáticos e testado para verificação da proposta, nesta etapa também é importante verificar a dependência de dados. Após isso é possível criar a micro-arquitetura do co-processador que inclui o pipeline, controle e paralelismo. E com todas as análises anteriores, os blocos estarão prontos para serem codificados em Verilog e o RTL poderá ser gerado.

3.3 Implementação e resultados

A implementação do circuito no hardware reconfigurável também pode ser dividida em três partes:

- Implementação.
- Teste funcional.
- Análise de performance.

As células do chip Cyclone-V-SoC do kit DE0-Nano-SoC são configuradas para executar os circuitos digitais feitos, configurando os barramentos e o sistema no geral de acordo com a análise da primeira parte. E a HPS com o processador ARM será programada em C.

A verificação do funcionamento do sistema será feita mediante testes os quais consistem em introduzir uma entrada conhecida no co-processador e verificar se a saída condiz com o resultado esperado. A análise de performance será feita de forma simples, analisando o tempo de execução com o co-processador e sem ele.

4 Desenvolvimento

4.1 Análise de custo computacional do UKF

O perfil do algoritmo UKF em relação ao seu custo computacional foi traçado para determinar as partes mais críticas do algoritmo com elevado custo computacional e identificar o gargalo do sistema para ser acelerado, levando em consideração o tempo, área e reutilização do algoritmo. Uma implementação em MATLAB do UKF foi feita seguindo o algoritmo e código criado por Dr.Yi Cao da Universidade de Cranfield (CAO, 2008). O código foi modificado para não utilizar a bibliotecas muito avançadas do MATLAB e estados foram adicionados baseando em sistemas VANTs encontrados como o projeto "PX4 autopilot". O projeto é um software de autopilotagem *open-source* para VANTs e possui 32 estados (PX4-AUTOPILOT, 11/2018), então foi definido que a aplicação de referência será de até 32.

O resultado no MATLAB e executando em um processador i7 está na imagem 3.

Função:	Chamadas	Tempo próprio (s)	Tempo p. total(%)	Tempo total (s)	Tempo total(%)
Exemplo	1	0.007	21.21%	0.033	100%
ukf	1	0.004	12.12%	0.025	75.76%
SigmasPoints	1	0.001	3.03%	0.012	36.36%
Cholesky	1	0.011	33.33%	0.011	33.33%
UT	2	0.005	15.15%	0.009	27.27%
Equação não linear	33	0.003	9.09%	0.003	9.09%
Equação de medição	33	0.001	3.03%	0.001	3.03%

Figura 3 – Perfil do UKF no MATLAB

A partir desse perfil é possível identificar que são duas tarefas que exigem maior esforço computacional: UT (unscented transform) e Sigmas Points. O cálculo dos Sigmas points era esperado pois é a etapa que executa a decomposição de Cholesky que demora mais de 30% do tempo total, mas o cálculo da *unscented transform* foi uma surpresa e tem um custo maior do que o previsto ocupando 27% do tempo total. É importante salientar que as funções: equação não linear e equação de medição são determinadas pelo usuário de acordo com seu sistema. Ou seja, é específica para cada aplicação e esse tempo pode variar consideravelmente. As equações utilizadas foram seguindo o exemplo de Dr.Yi Cao da Universidade de Cranfield (CAO, 2008).

É evidente que esses resultados podem se alterar se utilizarmos o algoritmo processado em um sistema embarcado que utiliza outros processadores e arquitetura. Então, o código do MATLAB foi transferido para linguagem c para ser implementado no kit

DE0-Nano SoC. O Cyclone V SoC possui dois processadores Cortex ARM A9, o perfil foi levantado utilizando a '*performance monitor unit*' da ARM, que consiste em contadores de nanosegundos implementados em hardware e que são utilizados para obter com precisão o tempo de execução para calcular a performance do processador.

O resultado foi similar ao da figura 3, como pode ser visto na tabela 1. A diferença foi que a função UT demorou mais que o SigmaPoint.

Função	Tempo(ns)	Tempo (%)
SigmaPoints	4295240	26.27%
UT	6389576	39.08%
Outros	5664093	34.65%
UKF	16348909	100.00%

Tabela 1 – Perfil do UKF no matlab

É importante ressaltar que a função UT ao utilizar a ferramenta '*code generator*' se tornou muito grande e nada otimizada, já a função sigmapoints ficou mais enxuta e foi utilizado um algoritmo de Cholesky otimizado. Assim é plausível que essa diferença também seja causada pelo código implementado em c.

A função UT usa metade de seu tempo para calcular equações que variam para cada aplicação. Então, considerando que o co-processador será mais útil se ele for empregado para qualquer aplicação que utilize UKF e que o tempo do Cholesky é tão significante quanto o do UT, foi escolhido acelerar o Cholesky.

Além disso, foi definido que cada dado dos sensores será um floating point de 32-bits. E como foram previstos a utilização de até 32 estados, as matrizes de entrada e saída do Cholesky terão lados de até 32.

4.2 Arquitetura do sistema

Um sistema embarcado é composto de vários subsistemas e por isso é interessante a utilização de chips SoCs em que em um único chip são implementados vários subsistemas. Isso permite uma melhor performance já que a comunicação é mais rápida pois a latência é menor, uma vez que os periféricos se ligam diretamente com o barramento interno. Além disso, a área ocupada na placa é menor pois apenas um chip contempla múltiplas funcionalidades, consequentemente o custo diminui já que o custo de fabricação da PCB é diretamente proporcional a área ocupada.

Pensando nisso, é comum que os aceleradores como por exemplo um acelerador de criptografia, sejam implementados dentro de um SoC que possua uma unidade de processamento principal e memória. Assim a performance, o consumo e a área são otimizados.

Em outras palavras, o acelerador é um dos subsistemas dentro de um SoC, um bloco de IP dentro de um chip.

Os sistemas embarcados que podem utilizar o acelerador de UKF proposto nesse trabalho tem as mesmas características de se beneficiar como um subsistema de um SoC. Por isso o chip CycloneV-SoC foi escolhido para esse trabalho, ele contém *hard IPs* como processador ARM Cortex A9, cache e células de FPGA que serão usadas para implementar o projeto do acelerador de UKF. Assim é possível utilizar o co-processador como parte de um SoC, fazendo comunicação direto no barramento do chip. A figura 4 mostra uma visão geral do SoC.

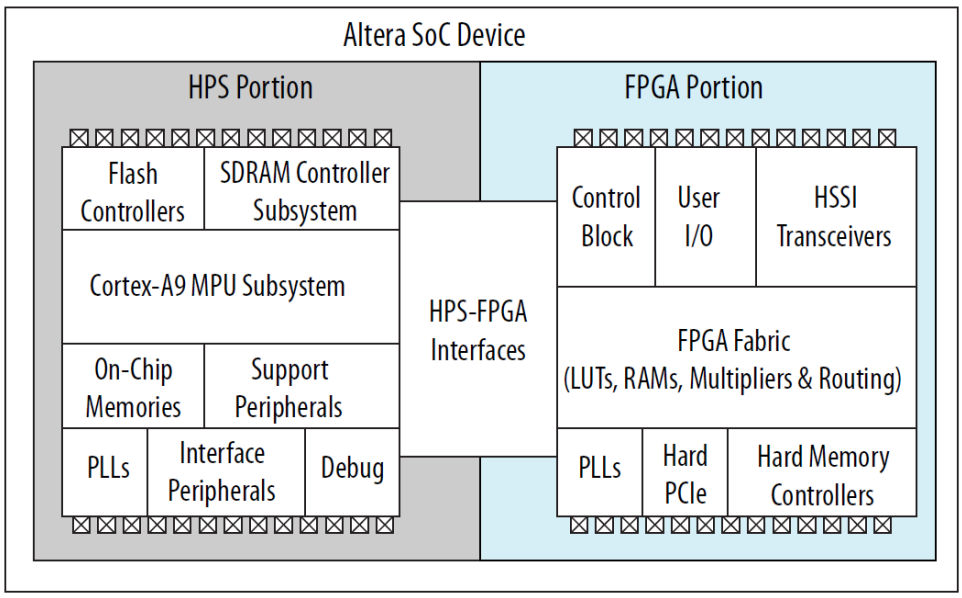

Figura 4 – Cyclone V SoC: Visão geral (INTEL, 6/2018, Cyclone V Design Guidelines)

O chip possui uma arquitetura incomum como pode ser visto na imagem anterior, portanto os desafios que esse capitulo busca resolver são: Entender a arquitetura de um SoC FPGA, os blocos disponíveis e recursos; Identificar as possíveis formas de transferência de dados entre a HPS e a FPGA ; discutir qual é o melhor fluxo de dados e determinar qual o *throughtput* entre a HPS e a FPGA.

4.2.1 Estudo da micro-arquitetura do chip Cyclone V SoC

O Cyclone V SoC não deve ser confundido com o popular chip Cyclone V que é apenas FPGA. Já Cyclone V SoC une microcontrolador e uma FPGA em um único chip, conforme a imagem 4 é possível ver então que o possui dois grandes blocos chamados de FPGA e HPS. A FPGA possui os blocos lógicos configuráveis e o HPS possui hard IPs (circuitos implementados fisicamente em silício).

Devido à arquitetura bem diferente dos chips populares, o estudo a seguir foi realizado objetivando entender os recursos do chip: blocos de hardwares disponíveis, características, barramentos, protocolos de comunicação e como implementar um acelerador de hardware. Os dados dessa seção em relação as características do chip foram retiradas dos documentos: Cyclone V Device Datasheet (INTEL, 05/2018), Cyclone V and Arria V SoC Device Design Guidelines (INTEL, 6/2018) e Cyclone V Hard Processor System Technical Reference Manual (INTEL, 1/2018).

Na imagem 6 é exibido de forma detalhada a arquitetura de topo.

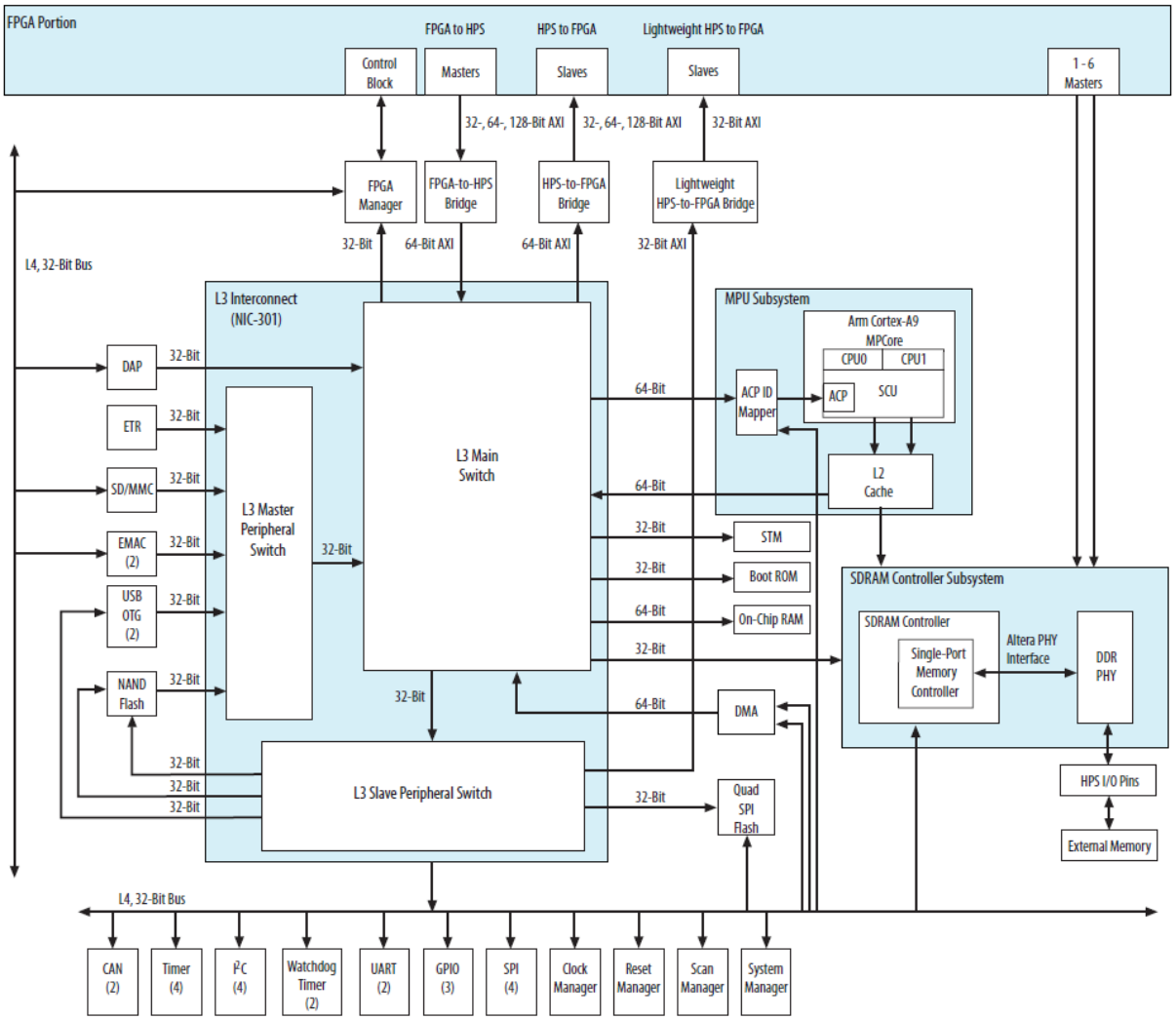

Figura 5 – Cyclone V SoC: Arquitetura de Topo (INTEL, 6/2018, Reference Manual)

Dentro da HPS é possível observar os seguintes subsistemas:

- Microprocessor Unit Subsystem:
  Contém dois Arm Cortex A9, memória cache, porta ACP e SCU.

- SDRAM Controller
  Permite a comunicação com uma memória RAM externa.

- Memórias e controladores:
  On-Chip RAM, Boot ROM e DMA.

- Periféricos de comunicação:
  Possui uma série de periféricos como timers, uart, spi, can, i2c.

Além disso blocos como L3 e L4 chamam a atenção por serem grandes, mas são apenas são *switches* para conexões entre os blocos.

Em relação a comunicação entre FPGA e HPS, é possível identificar três *bridges* entre o HPS e a FPGA e uma para a SDRAM externa.

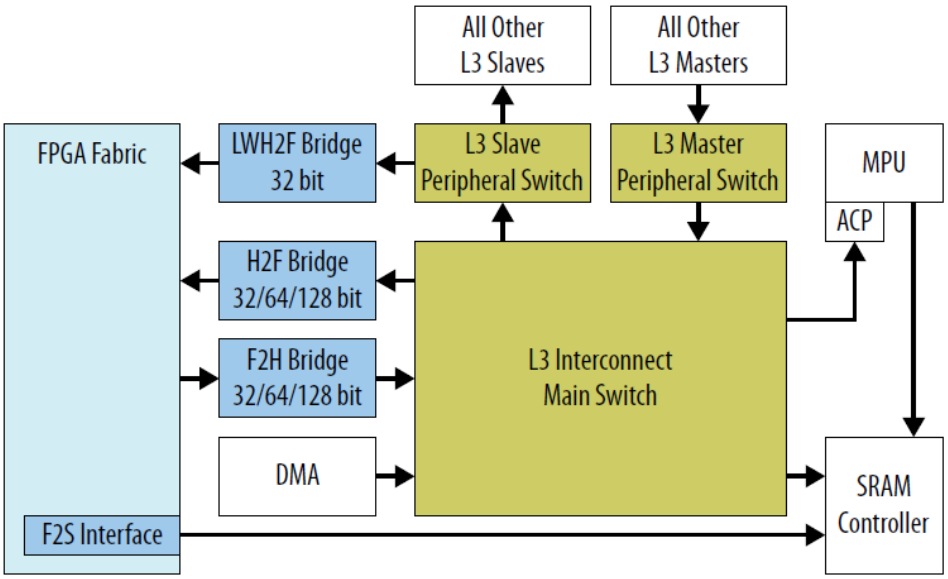

Figura 6 – Cyclone V SoC: Bridges (INTEL, 6/2018, Reference Manual)

- Lightweight HPS-to-FPGA Brigde ou LW-H2F:
  Permite que os mestres do lado da HPS acessem os mapas de memória do lado da FPGA. A MPU acessa essa ponte para controlar e verificar status de registrados. Como o acesso é apenas para esse tipo de informação possui um barramento limitado a 32 bits e com frequência máxima de 200 MHz, então não é recomendado o uso dessa porta para acesso a memória ou transferência de dados.

- FPGA-to-HPS ou F2H:
  Comunicação entre a FPGA e o HPS em que a FPGA é mestre. Ou seja, ela comanda o barramento. Essa *bridge* pode ser configurada com um *bandwidth* de 32, 64 ou 128 bits e permite um clock de 200 MHz. Essa *bridge* é utilizada para grande transferência de dados e acesso da FPGA a ACP (*accelerator coherency port*).

- HPS-to-FPGA ou H2F:
  Comunicação entre a HPS e a FPGA em que o HPS é mestre, blocos como MPU e DMA. Essa *bridge*, assim como a F2H, pode ser configurada com um *bandwidth* de 32,64 ou 128 bits e permite um clock de 200 MHz. Esta bridge é destinada para transferência de grandes quantidades de dados e servem tanto para escrita quanto para leitura de dados na FPGA.

- FPGA-SDRAM:
  Permite a comunicação direta da FPGA com a memória SDRAM externa. Podem ser utilizadas até seis portas de 64-bits sendo no máximo quatro para leitura ou escrita.

Toda comunicação é feita através do barramento AXI3. Esse barramento tem as seguintes características:

- Protocolo AMBA, criado pela empresa ARM.
- 5 Canais: Read Address, Read Data, Write address, Write Data, Write response.
- Permite leitura e escrita simultânea.
- Permite modo *burst*.
- Multi-mestre.
- Permite diferentes clocks ou tamanho para cada bloco conectado no barramento.

A tabela 2 o *throughput* teórico dos barramentos de comunicação. É dado como teórico pois é o máximo que o barramento consegue fornecer, em uma aplicação existem fatores que fazem com que a taxa diminua, como por exemplo: outros periféricos requisitando o barramento, ou o *miss* na cache.

Brigde	Largura do barramento	Clock Máximo	*Throughput* Teórico
H2F	128 bits	200MHz	51Gbps*
F2H	128 bits	200MHz	51Gbps*
LW-H2F	32 bits	200MHz	12.75 Gbps*
FPGA-SDRAM	256 bits	200MHz	102 Gbps*

Tabela 2 – Cyclone V SoC: *Throughput* dos barramentos

* Utilizando todos canais da AXI(Leitura e escrita simultâneas).

Um recurso interessante que o chip Cyclone V SoC possui, é a porta ACP. Ela permite o acesso direto da FPGA a cache da MPU, podendo utilizar o dado mais atual e

recém processado pela CPU. Isso é importante pois, ao processar dados, a CPU armazena os resultados mais atuais na cache, então se a FPGA desejar utilizar esse dado é necessário que a CPU pare o que estiver processando e copie esse dado para a memória e depois disso a FPGA pode acessar, consumindo muito tempo. Com a porta ACP esse problema é solucionado, acessando diretamente a memória cache.

Além desses recursos apresentados existem outros periféricos que são importantes na implementação de um sistema embarcado, como comunicação serial. Esse chip é muito útil para prototipagem pois ele permite a flexibilidade de uma FPGA ao mesmo tempo incorpora *hard-IPs* que estão presentes em SoCs.

4.2.2 Análise e discussão dos métodos de transferência de dados

Com o entendimento da arquitetura do chip, é possível determinar um fluxo de transferência de dados entre o lado do HPS e o co-processador na FPGA. Existem várias metodologias possíveis e cada uma tem as suas vantagens/desvantagens de acordo com a quantidade de dados a serem transferidos e onde eles se encontram. Foram observadas e analisadas as seguintes metodologias:

1. CPU escreve e lê na FPGA
   Comunicação direta do core (ARM A9) para a FPGA. O processador procura os dados requisitados na memória e envia pela HPS-to-FPGA. Posteriormente quando os cálculos estiverem prontos, a CPU é informada através de um sinal de interrupção gerado pelo co-processador, e utiliza o canal *read* para ler os dados calculados.

   - Vantagem
     - Simplicidade
   - Desvantagem
     - CPU fica ocupada enquanto está enviando e lendo dados.
     - Processo de copiar dados não eficaz e lento para grandes quantidades de dados.

2. DMA copia os dados para FPGA e escreve resultados na HPS
   Comunicação indireta do processador com o co-processador. O processador ao identificar um cálculo que pode ser acelerado pelo co-processador deverá informar a dma para que ela copie os dados necessários do lado da HPS para a FPGA. E os resultados também podem ser copiados pela DMA da FPGA para o HPS. Leitura e escrita simultâneas e feita pela FPGA-to-HPS, ou seja, é possível ler e escrever ao mesmo tempo, isso permite que em matrizes grandes, é possível começar a escrever os primeiros resultados enquanto os últimos dados ainda estão sendo processados.

- Vantagem
  - CPU fica disponível para executar outras tarefas ou entrar em *sleep*(modo de economia de energia). Isso reduz o consumo ou aumenta a performance do sistema
  - A leitura e escrita simultânea aumenta a performance
- Desvantagem
  - Ocupa mais espaço na FPGA.
  - Aumenta a complexidade do controle de dados.

3. DMA copia os dados para FPGA e CPU lê os dados
   Comunicação indireta do processador com o co-processador, mas diferentemente do item anterior, os resultados são lidos pela CPU utilizando a HPS-to-FPGA quando ela achar necessário.

   - Vantagem
     - CPU fica disponível para executar outras tarefas.
     - Ocupa menos espaço na FPGA que o item 2. Pois os IPs de DMA são unidirecionais, então seriam necessários dois blocos de DMA.
     - Diminui a complexidade do controle de dados em relação ao item 2.
   - Desvantagem
     - A leitura e escrita simultânea não é permitida.

4. DMA com ACP
   Comunicação indireta do processador com o co-processador. A DMA tem acesso a cache do processador, buscando o dado diretamente. Assim permite utilizar os dados mais recentes que foram manipulados pela CPU.

   - Vantagem
     - CPU fica livre para executar outras tarefas.
     - Melhor performance, pois não existe a necessidade da CPU ter que executar um flush na cache, visto que a DMA pode acessar diretamente a cache e copiar o dado requisitado, reduzindo o tempo.
   - Desvantagem
     - A DMA primeiramente checa se o dado requisitado está na memória cache, se não estiver procura na RAM. Então, Dependendo do tamanho dos dados transferidos e onde eles estão, pode ser uma armadilha utilizar a porta ACP pois aumentar o tempo de transferência visto que a DMA vai checar a cache sempre.

5. SDRAM para FPGA
   Comunicação indireta do processador com o co-processador. Os dados podem estar na SDRAM externa vindo de sensores por exemplo. Assim é possível utilizar a porta FPGA-to-SDRAM para acessar diretamente os dados. A CPU apenas informa que é necessário buscar os dados e em qual endereço via porta HPS-to-FPGA, posteriormente a DMA se encarrega da transferência.

   - Vantagem
     - CPU fica livre para executar outras tarefas.
     - Porta FPGA-to-SDRAM com barramento maior, aumentando o *throughput*.
   - Desvantagem
     - Baixo desempenho para pequenas quantidades de dados, pois a comunicação inicial entre o processador para o co-processador será relevante.
     - Apenas é interessante caso o dado já esteja na SDRAM. Caso esteja na memória interna é necessário que o processador transfira todos os dados para a memória externa, consumindo muito tempo.

6. Combinação de duas formas
   É possível combinar qualquer uma das formas para leitura ou escrita. Isso permite uma grande flexibilidade e pode ser interessante de acordo com a aplicação. Um exemplo: Os dados dos sensores são escritos diretamente na SDRAM, o co-processador trata esses dados e o processador vai utilizar os resultados do co-processador para tomada de decisão. Visto isso, seria interessante que a leitura fosse feita conforme o item 5 e a escrita fosse feita via DMA com ACP, pois os dados de leitura estão na SDRAM e o processador vai precisar utilizar os resultados e seria interessante economizar o tempo de busca na memória externa escrevendo diretamente em sua cache.

   - Vantagem
     - Melhora o desempenho expressivamente para a aplicação.
   - Desvantagem
     - Aumenta a complexidade.
     - Arquitetura específica de uma aplicação.

Outra importante definição sobre o sistema embarcado é a utilização de sistema operacional ou não. Segundo (MOLANES, 2018) existe uma queda de até 50% na performance se for utilizado um sistema operacional no chip Cyclone V SoC, por isso foi escolhido não utilizar um sistema operacional visando a performance.

Também em (MOLANES, 2018), são apresentados dados que são interessantes para a definição do fluxo de comunicação, pois faz uma análise de performance com resultados pertinentes para esse trabalho. Os dados foram obtidos a partir de um experimento simples em que se escrevia em um bloco de memória na FPGA ou lia do bloco. Os dados a seguir:

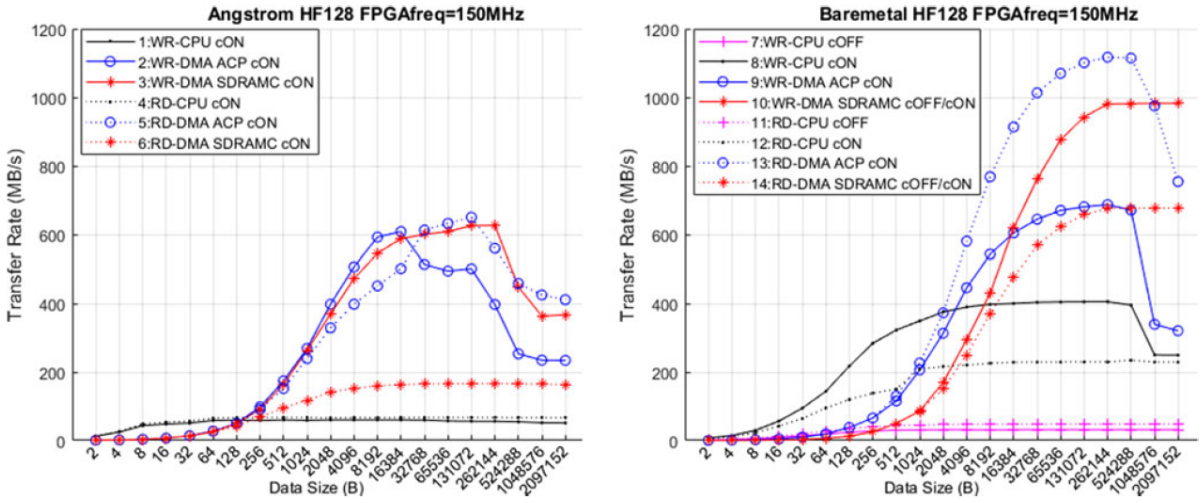

Figura 7 – Taxa de transferência entre HPS e FPGA do Cyclone V para diferente configurações (MOLANES, 2018)

O gráfico mostra o desempenho do chip usando o sistema operacional Angstrom (lado esquerdo) e sem o sistema operacional (lado direito). É possível ver que a medida que a quantidade de dados aumenta, a taxa de transferência varia e cada fluxo de transferência tem uma curva de *throughput* diferente. As siglas correspondem a:

- WR = Escrita
- RD = Leitura
- cOFF = Cache desligada
- cON = Cache ligada
- ACP = Porta ACP ligada
- DMA = DMA ligada
- SDRAM = Leitura ou escrita na memória externa

Esses resultados foram feitos de forma genérica e levando em consideração somente o *throughput*, mas existem outros fatores que influenciam como complexidade, quantidade de células utilizadas e onde o dado se encontra, esses fatores foram apresentados anteriormente. Assim, é possível determinar a melhor configuração analisando as vantagens/desvantagens de cada alternativa.

Pensando de forma prática no algoritmo unscented kalman filter, temos as seguintes conclusões: Os dados dos sensores como GPS, acelerômetro, giroscópio estarão armazenados na memória externa. O processador ao final do cálculo do UKF vai utilizar os dados para tomada de decisão. Por esse lado, a melhor combinação seria utilizar o item 5 (SDRAM-FPGA) para leitura e o item 4(DMA com ACP) para escrita. Mas o objetivo é acelerar apenas a parte mais crítica do algoritmo que foi definido como o cálculo da decomposição de Cholesky. No caso do Cholesky, o parâmetro recebido é a matriz de covariância que foi processada recentemente e a saída é a matriz que será utilizada para criar os *sigmas points*.

Então analisando o gráfico de performance, e os dados para a aplicações entre 2kB até 20 kB e levando em consideração a aplicação é preterível implementar o item 2(DMA copia os dados para FPGA e escreve resultados na HPS) ou o item 3(DMA copia os dados para FPGA e CPU lê os dados), mas ambos com ACP(verificar item 4). A escolha entre os dois será determinada na implementação, pois pode ser que não haja muitas células programáveis disponíveis após a criação do co-processador.

4.3 Co-processador

Nesta seção foi discutida a micro-arquitetura do co-processador. As definições e análises dos capítulos anteriores como o algoritmo a ser acelerado e o fluxo de dados entre a HPS e a FPGA, foram cruciais para criar uma micro-arquitetura que permita um paralelismo e pipeline.

4.3.1 Estrutura do algoritmo de Cholesky

A decomposição de Cholesky pode ser separada em duas equações:

$$L_j, j = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2} \tag{4.1}$$

$$L_j, j = \frac{1}{L_{i,j}}(A_{j,j} - \sum_{k=1}^{j-1} L_{i,k}\dot{L}_{j,k}) \tag{4.2}$$

A equação 4.1 resulta no valor da diagonal, já a equação 4.2 resulta na parte inferior da matriz. É possível ver que existem 4 tipos de operações: Multiplicação, Adição/Subtração, Divisão e raiz quadrada. E também que para calcular um elemento da matriz é necessário ter calculado anteriormente os elementos da mesma linha e colunas anteriores, além da diagonal da coluna que por sua vez também depende dos elementos anteriores.

Exemplificando:

$$\mathbf{A} = \begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{pmatrix}$$

Para $L = Cholesky(A)$, temos:

$$\mathbf{L} = \begin{pmatrix} \sqrt{A_{1,1}} & 0 & 0 & 0 \\ \frac{A_{2,1}}{L_{1,1}} & \sqrt{A_{2,2} - L_{2,1}^2} & 0 & 0 \\ \frac{A_{3,1}}{L_{1,1}} & \frac{A_{3,2} - L_{3,1}L_{2,1}}{L_{2,2}} & \sqrt{A_{3,3} - L_{3,1}^2 - L_{3,2}^2} & 0 \\ \frac{A_{4,1}}{L_{1,1}} & \frac{A_{4,2} - L_{4,1}L_{3,1}}{L_{2,2}} & \frac{A_{4,3} - L_{4,1}L_{3,1} - L_{4,2}L_{3,2}}{L_{2,2}} & \sqrt{A_{4,4} - L_{4,1}^2 - L_{4,2}^2 - L_{4,3}^2} \end{pmatrix}$$

Pelas equações 4.1, 4.2 e pelo exemplo acima é possível visualizar a dependência de dados em que para calcular um elemento é necessário ter calculado outros elementos anteriormente, exceto o $L_{1,1}$. A figura 8 ilustra a dependência de dados. Os elementos circulados são os analisados em relação a sua dependência, as setas apontam para o dado necessário.

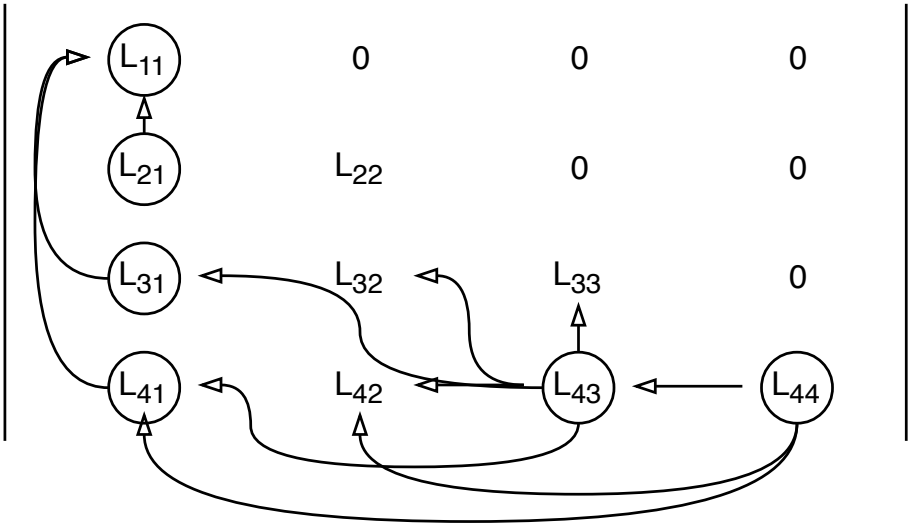

Figura 8 – Dependência de dados - Cholesky

A análise da dependência de dado é importante pois o co-processador precisará tratar isso, podemos classificar esse problema como um *hazard* de dados. Em computação, *hazard* de dados acontece quando uma instrução precisa de um dado que ainda não foi calculado pois está em um estágio do pipeline, ignorar isso torna o resultado incorreto pois o cálculo utilizará um dado não atual (PATTERSON, , 2009).

4.3.2 Blocos de hardware básicos para Cholesky

O algoritmo foi decomposto em blocos digitais matemáticos. A equação 4.1 determina a diagonal da matriz e ela foi quebrada conforme a representação dada pela figura 9.

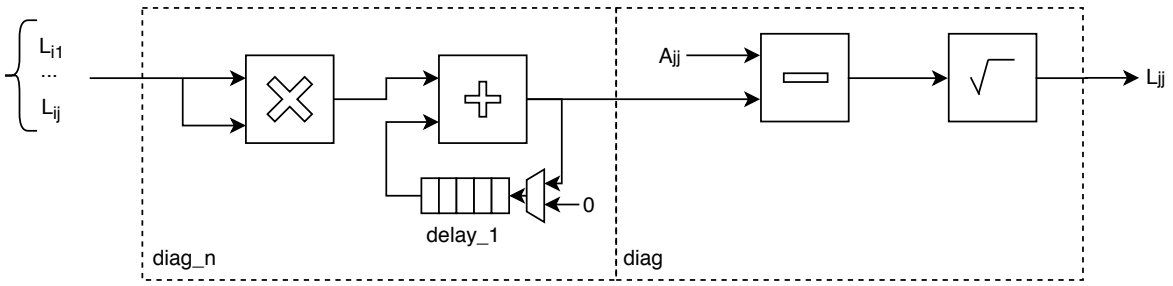

Figura 9 – Diagrama de blocos para diagonal do Cholesky

Os cálculos dos elementos da diagonal pode ser dividido em dois blocos:

- Diag_n: Cálculo do somatório da equação 4.1. Os dados que entram no bloco são os elementos já calculados da matriz de cholesky referente a linha da diagonal desejada.

Os elementos serão multiplicados por eles mesmos e feito um somatório com um bloco somador e um *loop* que conecta a saída na entrada do próprio bloco. A saída do multiplexador será zero para o primeiro elemento e posteriormente sua saída será a saída do somador. O delay_1, possibilita um atraso que deve ser ajustado para sincronizar os dados da entrada, foi colocado pensando em auxiliar no pipeline.

- Diag: Cálculo final do elemento da diagonal. Após o somatório ser calculado, o elemento A_{JJ} deve ser subtraído do somatório e a raiz desse resultado será o elemento da diagonal.

O *loop* no bloco diag_*n* é um problema pois gera uma latência em que o bloco diag fica esperando o resultado do somatório. Mas essa arquitetura foi feita já pensando em executar um pipeline, reutilizar resultados e paralelizar o que for possível. Isso será mostrado nas próximas seções.

Seguindo o raciocínio, a equação 4.2 foi quebrada conforme a representação da figura 10.

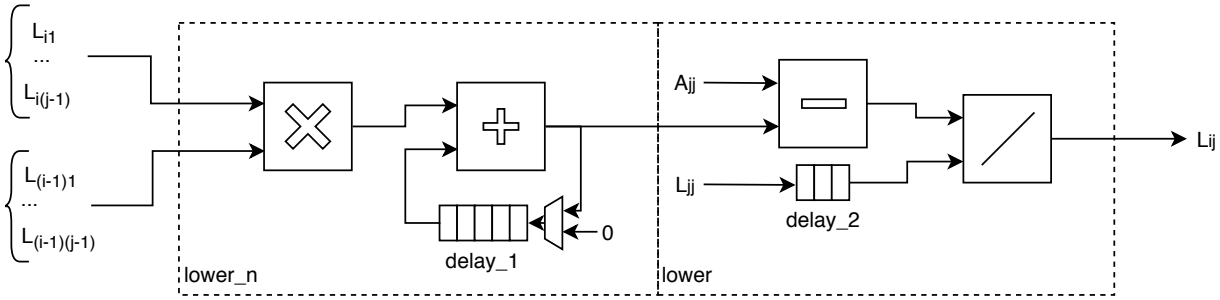

Figura 10 – Diagrama de blocos para elementos abaixo da diagonal do Cholesky

O cálculo dos elementos abaixo da diagonal também podem ser divididos em dois blocos:

- Lower_n: Cálculo do somatório da equação 4.2. Os dados que entram no bloco são os elementos já calculados da matriz de cholesky referente as colunas anteriores do elemento a ser calculado e sua linha, bem como a linha anterior. Os elementos de entrada serão multiplicados e o resultado é somado/acumulado com um bloco somador e um *loop* que conecta a saída na entrada do próprio bloco. A saída do multiplexador será zero para o primeiro elemento e posteriormente sua saída será a saída do somador.O delay_1 e o delay_2 possibilitam um atraso que deve ser ajustado para sincronizar os dados da entrada, foi colocado pensando em auxiliar no pipeline.

- Diag: Cálculo final do elemento. Após o somatório ser calculado, o elemento A_{ij} deve ser subtraído do somatório e dividido pelo elemento da diagonal de sua coluna.

Assim como na figura 9, o *loop* na primeira parte causa uma latência que foi resolvida com o pipeline, reutilização de resultados e paralelismo de dados que será discutido a seguir.

4.3.3 Micro-arquitetura: Pipeline, Paralelismo e reutilização de dados

Nesta subseção serão apresentados e discutidos as micro-arquitetura do co-processador. Para obter performance, foram arquitetados a reutilização de resultados, um pipeline e processamento de dados em paralelo.

Pelo exemplo na subsecção 4.3.1 e pela equação 4.2 fica fácil visualizar que dado um elemento da matriz em que i>j e j>2, o próximo a sua direita utiliza dados já calculados. Por exemplo o elemento $L_{4,2}$ calcula $L_{3,1} \cdot L_{3,1}$, que também é necessário no elemento $L_{4,3}$. Além disso, os elementos da diagonal precisam do quadrado de todos os elementos anteriores da mesma linha, conforme a equação 4.1, assim é interessante executar o quadrado deles assim forem calculados. Isso permite que ao executar o cálculo da diagonal, todos os quadrados já foram calculados diminuindo a latência do *loop*.

Portanto, juntando os blocos 10 e 9, acrescentando a reutilização os dados já calculados juntamente com alguns blocos de atraso para auxiliar no fluxo de dados e executando o quadrado de cada elemento logo após o cálculo do mesmo, temos a seguinte arquitetura:

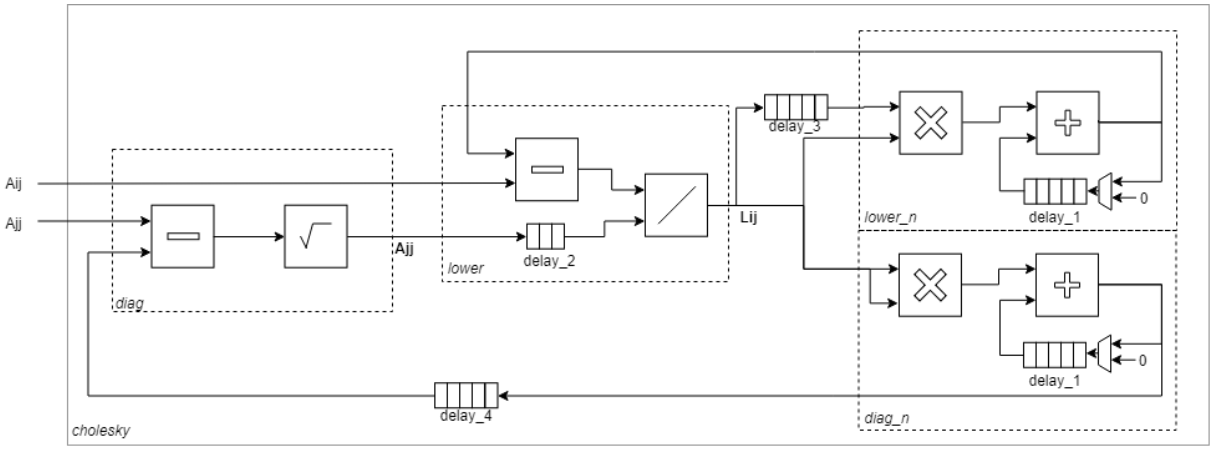

Figura 11 – Diagrama de blocos do Cholesky sem controle

O diagrama da figura 11 é interessante pois mostra o algoritmo como um todo. Mas ainda existem elementos pendentes como blocos de controle e pipeline que serão discutidos na seção a seguir.

Os blocos diag_n e lower_n tem os mesmos elementos, mas eles foram duplicados pois é possível paralelizar. Se for utilizado um só bloco, será necessário executa-lo duas vezes, assim aumenta o tempo e diminui a performance. A custo disso é o aumento de área/células configuráveis.

Na figura 11 é fácil identificar três etapas do algoritmo: Cálculo da diagonal, cálculo do elemento abaixo da diagonal e cálculo das variáveis necessária para resolução dos elementos das próximas colunas. Essas etapas foram definidas como as etapas do pipeline.

A subseção 4.3.1 discute sobre a dependência de dados e é observável pelas equações 4.1 e 4.2, além da figura 8 que os elementos abaixo da diagonal principal de uma mesma coluna são dependentes entre si, então é possível paralelizar seu processamento.

Objetivando melhorar o desempenho foram arquitetados então o pipeline e replicação de blocos para o paralelismo, que vai impactar na redução do tempo total. A figura 12, mostra a implementação simplificando a parte do controle e a saída de dados.

Os blocos lower, lower_n e diag_n foram repetidos de forma a processar 4 elementos em paralelo, pois o barramento permite a entrega de 128 bits, o que equivale a 4 palavras de 32 bits. Assim, é possível processar os 128 bits de uma vez se a diagonal da coluna já estiver sido calculada.

A arquitetura também foi feita pensando no vetor de dados de entrada. Os dados de entrada serão enviados da seguinte maneira: Primeiramente todas as diagonais serão enviadas e posteriormente os dados de cada linha por colunas serão enviados de 4 em 4 pois o barramento é de 128 bits.

Um problema que essa estrutura apresenta é que dependência de dados devida a natureza do algoritmo de cholesky e o cálculo da raiz quadrada que é demorado, causa uma latência muito grande entre o termino do cálculo da diagonal(bloco diag) e o início dos cálculos dos elementos da linha(bloco lower). Isso causa um problema no fluxo de dados de entrada, pois enquanto o bloco diag é processado os dados de entrada do bloco lower vão continuar vindo nesse tempo mas não poderão ser processados. Ou seja, mais dados serão entregues do que são processados nesse momento. Então é necessário um controle dos dados de entrada com um buffer, para tratar os momentos em que entram mais dados do que é processado. Existem também momentos em que são processados mais dados do que entram, se o clock do co-processador for maior que o do barramento. A solução para esse problema foi implementar um bloco de controle com uma FIFO.

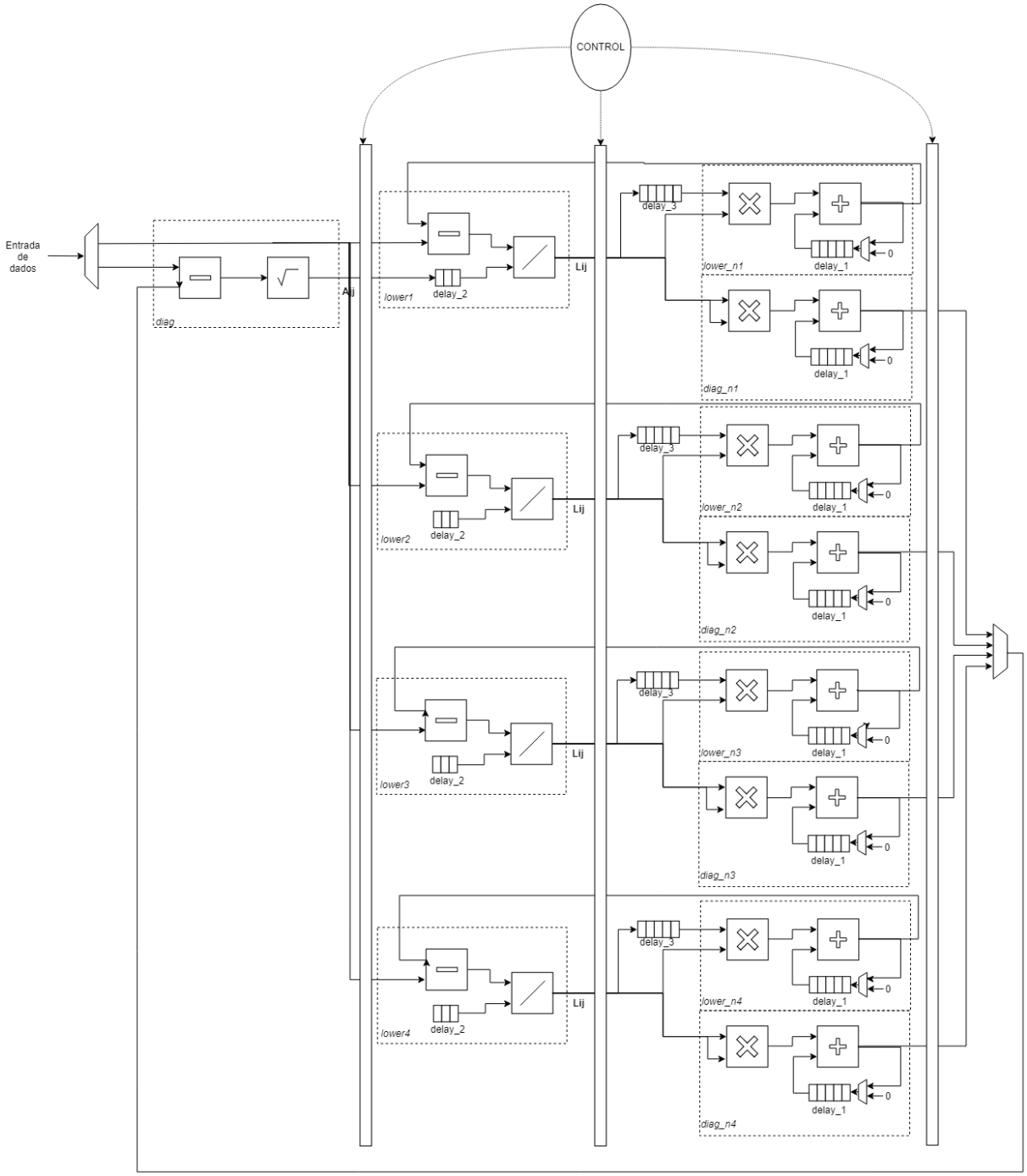

Figura 12 – Arquitetura de topo do Co-processador

Em relação ao pipeline, a figura 13 ilustra um exemplo de pipeline para uma matriz de tamanho 13. No primeiro ciclo é calculado o elemento $A_{1,1}$. No segundo ciclo é executado o cálculo de quatro elementos abaixo da diagonal em paralelo. No terceiro ciclo calculado em paralelo quatro elementos abaixo da diagonal e também dados necessários para calcular os elementos da próxima coluna. O pipeline segue esse raciocínio, mas na linha 14 o bloco lower1 não é executado pois seu resultado será nulo.

	1	2	3	4	5	6	7
1	DIAG(A 1,1)						
2		LOWER(A 2,1)	lower_n (A 2,1) diag_n (A 2,1)	DIAG(A 2,2)			
3		LOWER(A 3,1)	lower_n (A 3,1) diag_n (A 3,1)				
4		LOWER(A 4,1)	lower_n (A 4,1) diag_n (A 4,1)				
5		LOWER(A 5,1)	lower_n (A 5,1) diag_n (A 5,1)				
6			LOWER(A 6,1)	lower_n (A 6,1) diag_n (A 6,1)			
7			LOWER(A 7,1)	lower_n (A 7,1) diag_n (A 7,1)			
8			LOWER(A 8,1)	lower_n (A 8,1) diag_n (A 8,1)			
9			LOWER(A 9,1)	lower_n (A 9,1) diag_n (A 9,1)			
10				LOWER(A 10,1)	lower_n (A 10,1) diag_n (A 10,1)		
11				LOWER(A 11,1)	lower_n (A 11,1) diag_n (A 11,1)		
12				LOWER(A 12,1)	lower_n (A 12,1) diag_n (A 12,1)		
13				LOWER(A 13,1)	lower_n (A 13,1) diag_n (A 13,1)		
14					-	-	-
15					LOWER(A 3,2)	lower_n (A 3,2) diag_n (A 3,2)	DIAG(A 3,3)
16					LOWER(A 4,2)	lower_n (A 4,2) diag_n (A 4,2)	
17					LOWER(A 5,2)	lower_n (A 5,2) diag_n (A 5,2)	

Figura 13 – Pipeline

4.3.4 Blocos de controle

Os blocos de controle possuem objetivos bem claros, mas sua implementação é complexa. Os blocos de controle precisam tratar:

a) Aquisição dos dados de entrada:
É necessário buscar os dados da matriz de entrada. Isso será feito por uma DMA, conforme discutido no capitulo 4.2.2.

b) Fluxo de entrada de dados:
O processamento dos dados que entram não é homogêneo, é necessário um controle e um buffer para quando os dados entrarem mais rápido do que são processados, e para quando os dados são processados mais rápidos do que entram.

c) Fluxo de saída de dados:
Os resultados precisam ser reescritos na memória, e para isso é necessário um controle desses dados. Pois não será em todo ciclo de clock que os dados estarão

disponíveis, é preciso uma lógica que identifique os elementos prontos, agrupe e escreva na memória. Além da escrita da *flag* que mostra para o processador principal que o cálculo terminou.

d) *Hazard* de dados (internos e externos):
Mesmo que exista um buffer/controle de entrada de dados, podem existir instantes em que o processador requisite dados mas o buffer está vazio. Isso pode acontecer quando o clock do co-processador for maior que o do barramento, ou quando o número de elementos da matriz é muito grande, pois o co-processador recebe todas as diagonais primeiro. Além disso, existem os *hazards* de dados que já foram mencionados. Acontece quando é requisitado um resultado para calcular o próximo elemento, mas esse dado não está pronto, por exemplo: o bloco lower está pronto para começar o cálculo mas precisa do resultado do diag que não terminou, ou a mesma situação do bloco diag com o diag_n. Então é necessário tratar esse problema parando o pipeline até que o dado termine.

e) Fluxo de dados internos do co-processador:
Existem multiplexadores e demultiplexadores dentro do bloco principal do co-processador, por exemplo o multiplexador que determina qual dos diag_n será a entrada do subtrator do bloco diag. É preciso uma lógica de controle para eles. Além disso é necessária uma lógica para determinar se todos os dados de entrada foram processados e o cálculo terminou.

f) Validação dos dados:
Existem dados inválidos que serão calculados pela lógica combinacional. É preciso um controle de quais dados realmente são validos e em qual estágio do pipeline está o dado.

g) Determinar tamanho da matriz:
O co-processador deve ser genérico. O primeiro dado enviado corresponde ao tamanho da matriz. Assim, é necessário um tratamento desse primeiro dado, informando a outros blocos de controle sobre o tamanho da matriz e quantos *loops* serão necessário.

Não é necessário a criação de um bloco de controle para cada situação apresentada. Um bloco de controle pode ter mais de uma das funcionalidades descritas.

4.4 Implementação

Utilizando os softwares Quartus II, QSYS, modelSIM e Intel EDS-SoC foi feita a implementação de todo o projeto. Nessa seção serão apresentados o fluxo da parte pratica do trabalho, o sistema do SoC e os blocos do co-processador implementados em verilog.

Após todas as definições vistas nas seções anteriores, o projeto foi implementado na pratica utilizando o kit de desenvolvimento DE0-Nano-SoC mostrado na figura 14.

Como o projeto mistura hardware e software em um único chip, seu desenvolvimento é diferente de um microcontrolador convencional ou de uma FPGA, incluindo a gravação no chip que não é trivial como de um microcontrolador por exemplo.

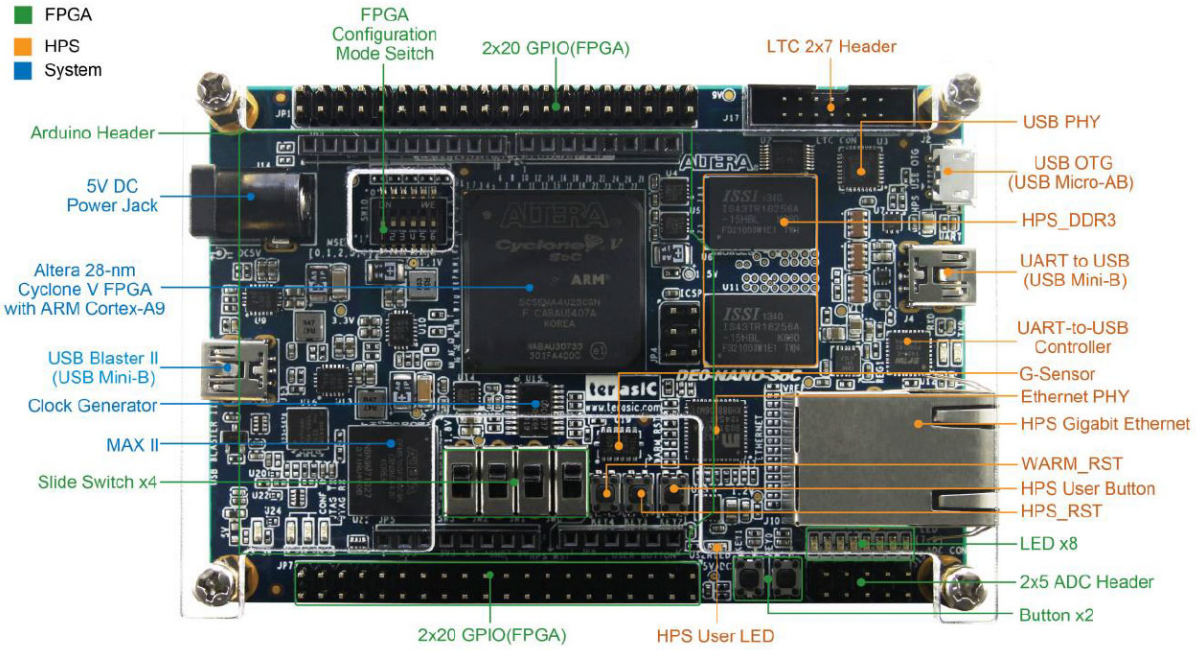

Figura 14 – DE0-Nano-SoC (TERASIC, 11/2018)

4.4.1 Fluxo do Projeto

As etapas de desenvolvimento no kit DEO-Nano-SoC sem entrar em detalhes foram descritas a seguir para ilustrar o processo feito nesse trabalho. O fluxo refere-se a uma aplicação *baremetal*(sem utilizar sistema operacional) e vai ser diferente se for utilizado um sistema operacional, pois o desenvolvimento do software e de gravação no chip serão diferentes das descritas.

- Projeto do hardware

1. Executar o *DE0-Nano-SoC System Builder* encontrado no site da Terasic (TERASIC, 11/2018) e criar um novo projeto selecionando os periféricos do kit que deseja utilizar.
2. Criar um projeto no software Quartus II.
3. Criar um projeto no QSYS.
   QSYS é uma ferramenta dentro do Quartus II que facilita a criação de SoCs.
4. Projetar no QSYS o sistema.
   A parte de hardware da HPS pode ser configurada no QSYS adicionando o IP *Hard Processor System* e também é possível adicionar outros blocos de IPs para a interface da HPS com a FPGA, como DMA e memória.
5. Gerar o projeto SoC.
   O QSYS gera vários arquivos, incluindo '.v' com os IPs selecionados, '.html' mostrando a composição do sistema e outros arquivos de projeto.
6. No projeto do Quartus II, adicionar os blocos.
   Nessa etapa o sistema já está configurado e os blocos devem ser adicionados. No caso desse projeto o bloco de topo do co-processador e conecta-lo ao bloco chamado SoC gerado pelo QSYS.
7. Compilar o projeto do Quartus, gerando o arquivo .SOF
8. Converter o arquivo .SOF para .RBF

- Projeto do software
  O software embarcado foi feito utilizando um editor de texto e o software EDS-SoC da intel. É possível utilizar o software ARM-DS5 que permite criar aplicações baremetal e depurar, mas é necessária licença.

  1. Verificar as bibliotecas disponíveis na HWLIB fornecida pela intel.
     A biblioteca facilita muito a programação em baremetal. Foi utilizada transferência de dados utilizando a DMA e habilitação da cache e porta ACP.
  2. Mapear periféricos e IPs.
     No projeto do QSYS é possível ver o endereço da HPS e dos periféricos que são necessários para programar.
  3. Criar a aplicação e compilar utilizando o software EDS - SOC.

- Gravação e execução

  1. Formatar o cartão SD com 3 partições com diferentes formatos de acordo com o documento: Cyclone V Design Guidelines (INTEL, 6/2018).
  2. Criar preloader a partir do software EDS - SoC.

3. Criar U-Boot.
4. Compilar o projeto.
5. Copiar o arquivo .bin, .rbf, uboot e o preloader gerado para partição FAT do cartao SD.
6. Utilizar interface serial no computador que vai estar conectada a porta serial UART do kit de desenvolvimento. Utilizei o software PUTTY (PUTTY, 11/2018).

4.4.2 Implementação do sistema de transferência de dados

Primeiro passo da parte prática consistiu em elaborar e implementar o sistema de transferência de dados que foi definido na seção 4.2.2. A HPS foi configurada de maneira a oferecer o melhor *throughput*. Essa configuração foi baseada no que foi discutido e em testes práticos. A imagem a seguir é do projeto no QSYS:

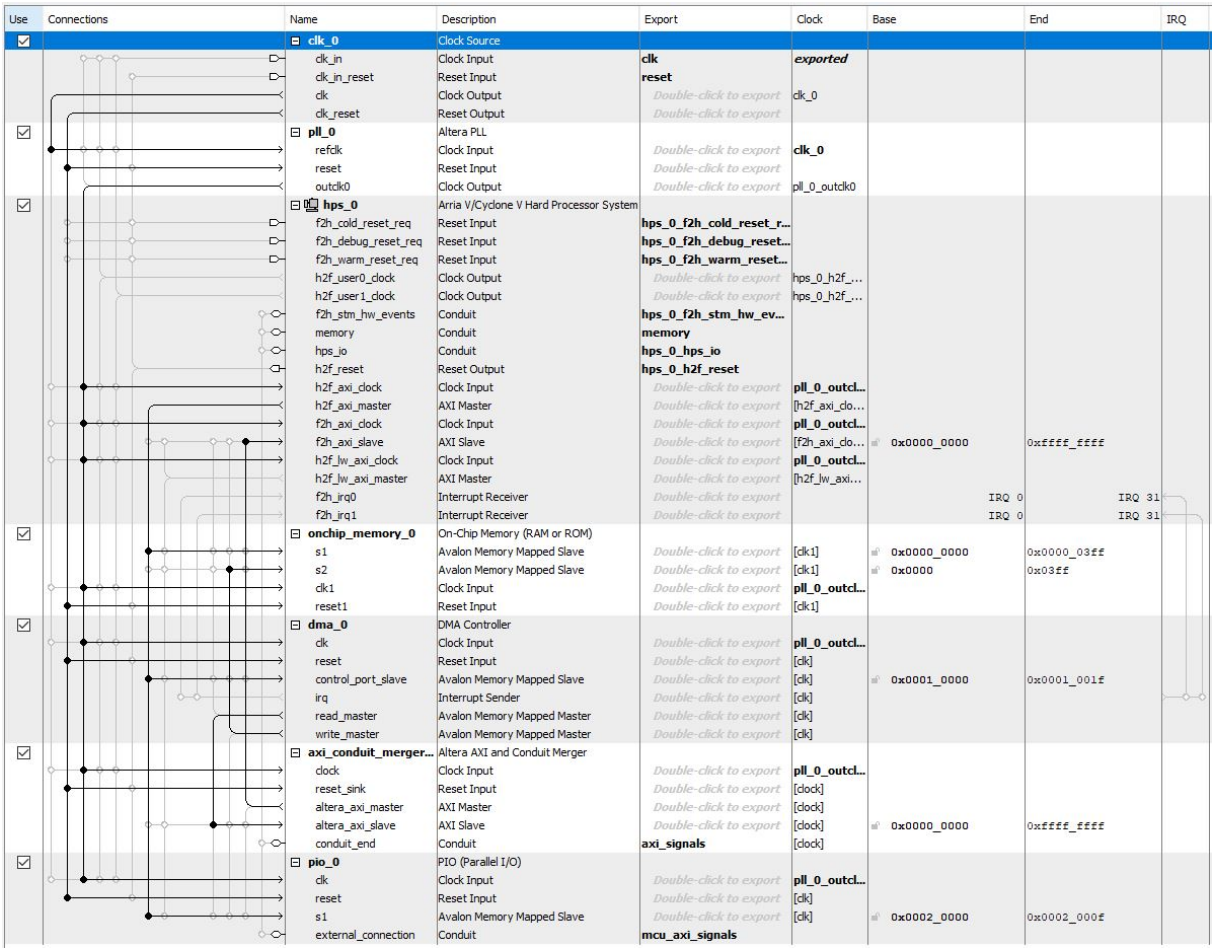

Figura 15 – SoC Sistema - QSYS

As configurações da HPS são: clock de até 150MHz no barramento, barramento de 128 bits para F2H e H2F, barramento de 32 bits para LW-H2F e 800MHz na MPU.

A parte do software embarcado foi feita em C e usando a biblioteca HWLIB para a transferência de dados via DMA, habilitar a cache e a porta ACP.

Testes para verificar o throughput real foram feitos. Para uma transferência de 2kB com um clock do barramento de 100MHz demorou 27941 nanosegundos, utilizando a *Performance Management Unit*.

4.4.3 Implementação do co-processador

O desenvolvimento do co-processador foi feito em verilog e usando o software Quartus II para gerar o rtl e arquivo '.SOF'. O projeto foi simulando com o software modelSIM. Foram desenvolvidos os blocos a seguir e estão disponível em https://github.com/acoimbramendes/hardwareaccelerator.

Blocos de Hardware desenvolvidos:

- Blocos matemáticos básicos: Diag, lower, ndiag e nlower.
  Os blocos matemáticos básicos das figuras 9 e 10 foram criados utilizando IPs de unidades aritméticas fornecidos pelo próprio software Quartus II. Existe uma biblioteca de unidades de *floating points*(pontos flutuantes) (INTEL, 12/2016, Floating-Point IP Cores User Guide). O problema desses IPs é que possuem pipeline de vários estágios conforme a tabela 3, principalmente a raiz quadrada. A vantagem é que permite um clock mais rápido do que se fosse implementado. Além disso foi feito um vetor que acompanha o número identificando onde no pipeline ele está.

Bloco	Estágios	Frequência Máxima
Adição e Subtração	7	≈230MHz
Multiplicação	5	≈300MHz
Divisão	6	≈300MHz
Raiz Quadrada	16	≈470MHz

Tabela 3 – Floating Point IPs

  O bloco de adição e subtração apresenta uma quantidade alta de estagios e uma frequência muito baixa pela simplicidade do bloco. Uma possível melhoria seria criar os próprios blocos e não utilizar os IPs do quartus. Isso não foi feito pois primeiramente foi focado o completo funcionamento do co-processador, no capitulo 5 foram discutidas as possíveis melhorias e se elas fazem sentido para o resultado final.

- cholesky:
  Integra os blocos matemáticos básicos conforme a figura 12 e os blocos de controle cholesky_control, stop_pipe, latch_a, control_ndiag e div_in.

- cholesky_control
  Através de logica combinacional e maquinas de estado controla o pipeline e o fluxo de dados internos. Gera os sinais para controle dos multiplexadores; controla em qual coluna da matriz o processador está; gera a *flag* que indica que os cálculos estão terminados; gera sinais que indicam onde os dados estão no pipeline; gera sinais que indicam que os dados estão prontos.

- stop_pipe
  Controla o pipeline em relação aos dados de entrada. Se a fifo estiver vazia e um dado for requisitado o pipeline para por alguns ciclos e em seguida é retomado. O clock é cortado.

- latch_a
  Apenas mantém o valor da diagonal da linha que está sendo processada. Atualiza quando uma nova linha começa e a diagonal é calculada.

- control_ndiag
  Controla por maquina de estado e logica combinacional qual valor entra no bloco diag para calcular a diagonal. Como existem 4 blocos ndiag, é necessário um controle de qual valor deverá entrar no bloco diag.

- div_in
  Bloco que apenas divide o número de lados da matriz de entrada pelo número de blocos lower em paralelo (definido como 4). A saída vai para o 'cholesky_control' que utiliza esse dado para distribuir os dados de entrada entre os quatro blocos lower.

- Bloco fifo_interface:
  Buffer de entrada. Composto de 5 blocos de fifo de 512 bits. Possui dois acessos simultâneos, o clock pode ser diferente para cada um dos lados.

- fifo_control:
  Controla e distribui os dados de entrada para as 5 fifos através de uma máquina de estado e logica combinacional. Além disso atribui o primeiro dado de entrada a div_in.

- output_control:
  Controla os dados de saída, verificando se o dado de entrada é válido e salvando na memória integrada.

- ukf_top:
  Bloco de topo do co-processador, integrando o fifo_interface, fifo_control, output_control e cholesky.

  Foram criados vários arquivos de testes para simular os blocos foram separadamente e juntos.

4.4.4 Integração do co-processador no sistema SoC

O QSYS da imagem 15 foi levemente alterado, liberando a saída da DMA e a entrada S2 da memória. Assim no bloco de topo do SoC foi conectado a saída da DMA no ukf_top para fornecer os dados de entrada, e a saída do ukf_top se conecta a memória para fornecer os resultados.

A figura ilustra de forma geral como os blocos interagem. Os blocos DMA e memory/DMA 2, representam IPs que foram utilizados e levemente modificados.

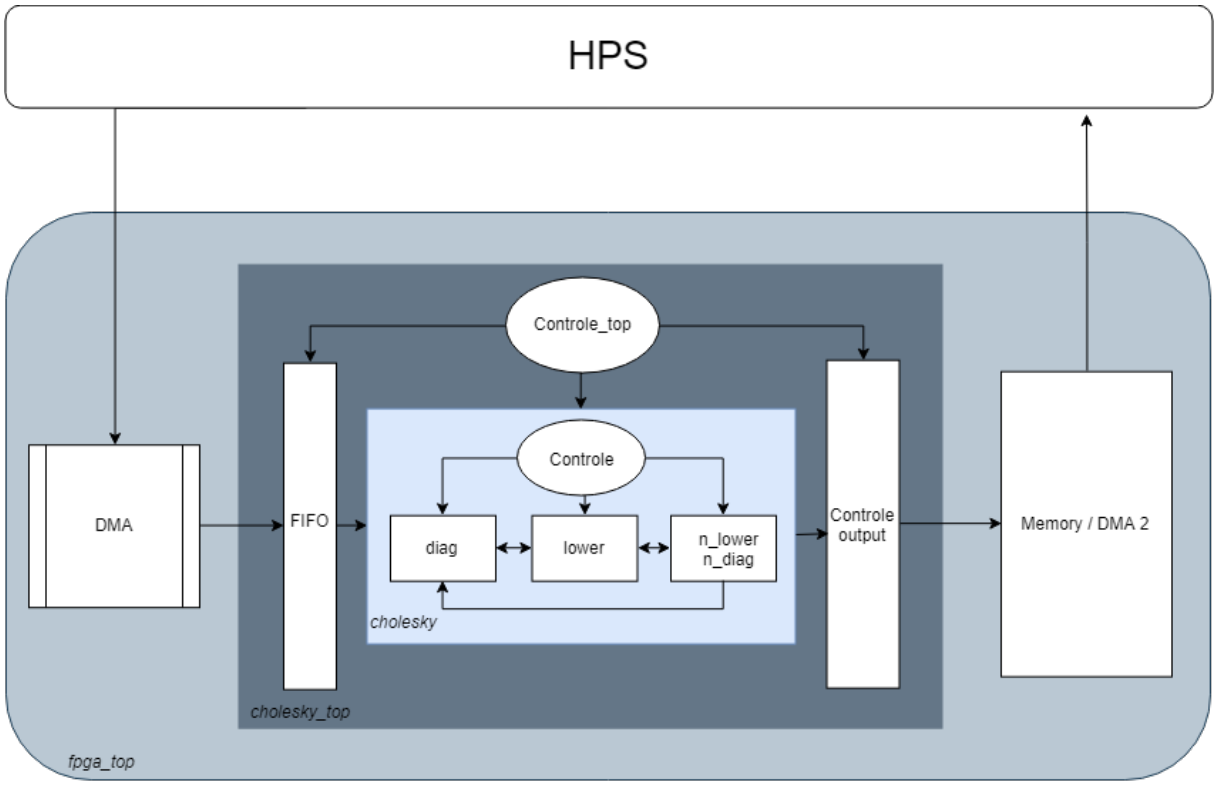

Figura 16 – Diagrama de topo

O fluxo de todo o processamento é o seguinte:

1. Processador começa o cálculo do UKF.
2. Processador chama a função Sigma Points.
3. A função sigma points configura a DMA para a transferência dos dados, passando um ponteiro para os dados da matriz de entrada e o tamanho da matriz.
4. A DMA é responsável por requisitar a utilização do barramento e transferir os dados. A partir dessa etapa o processador principal está livre para executar outras tarefas.
5. A fifo ao receber o primeiro dado (tamanho da matriz), informa ao bloco de controle (controle_top) que um cálculo está sendo requisitado e qual o tamanho da matriz.
6. Os primeiros dados estarão sendo enviados e armazenados nas FIFOs pela DMA, sendo distribuídos de acordo com o bloco de controle, identificando se é um dado referente a diagonal e qual raia ele deve entrar. Em paralelo a isso, o bloco Cholesky já começou a calcular os primeiros dados da matriz resultante.
7. A medida que os elementos da matriz resultante forem sendo calculados, o bloco control_output envia esses dados para a memória/DMA2.
8. A matriz resultante é calculada de coluna em coluna.
9. Quando todas as colunas forem processadas, o bloco controle informa aos demais blocos de controle que o calculo terminou. Assim, o bloco controle_output escreve na *flag* que indica para a HPS que os dados estão disponíveis.
10. O processador principal continua o calculo do UKF com o resultado do co-processador.

4.5 Resultados e discussão

Os resultados da implementação do sistema e do co-processador estão a seguir:

1. Teste da transferência de dados:

   O teste consistiu em transferir os dados para a FPGA utilizando a *bridge* F2H e uma DMA. Posteriormente os dados foram lidos utilizando a *bridge* H2F e foi verificado se a transferência foi um sucesso.

Tamanho dos dados	Clock	Tempo
1 kB	100MHz	15082 ns
2 kB	100MHz	27941 ns
4 kB	100MHz	54358 ns
8 kB	100MHz	108236 ns
16 kB	100MHz	214698 ns

Tabela 4 – Performance da Transferência de dados

2. Teste do Cholesky:

   O Cholesky foi testado e o resultado foi verificado utilizando o MATLAB como referência.

Hardware	Tamanho da matriz	Clock	Tempo	Precisão
Dual ARM A9	32	800MHz	3410795 ns	Dupla[3]
Co-processador[1]	32	100MHz	156950 ns	Simples[4]
Co-processador[2]	32	100MHz	161901 ns	Simples[4]

[1] Simulado
[2] Implementado no kit DE0-Nano-SoC
[3] 32 bits com 23 bits de mantissa, equivalente a até 7 dígitos decimais.
[4] 64 bits com 53 de mantissa, equivalente a até 15 dígitos decimais.

Tabela 5 – Resultados de performance do cholesky

3. UKF:

Hardware	Estados	Tempo
Somente no Processador	32	16348909 ns
Processador + Co-processador*	32	13095064 ns

*Estimado

Tabela 6 – Resultado de performance do UKF

4. Recursos da FPGA utilizado:

    a) Blocos lógicos: 8,153 / 15,880 (51%)

    b) Total de Registadores: 9213

    c) Total de pinos: 231 / 314 (74%)

    d) Total de blocos de memória: 243,355 / 2,764,800 (9%)

    e) Total de blocos de DSP: 32 / 84 (38%)

    f) Total de PLLs: 1 / 5 (20%)

    g) Total de DLLs: 1 / 4 (25%)

Pelos resultados discriminados na tabela 5 é possível ver que o co-processador com uma frequência de 100MHz conseguiu uma performance bem acima do esperado, pois conseguiu ser 21,06 vezes mais rápido do o processamento do cholesky no Dual ARM A9 do Cyclone-V-SoC com um clock oito vezes mais rápido (800MHz). O que representa uma redução de tempo em 95,25%. Esse resultado é devido a arquitetura de processamento de dados do tipo *floating point* em paralelo com o pipeline e reutilização de dados calculados que foi descrita em 4.3.

Pela performance inesperada, uma biblioteca encontrada no github e desenvolvida por Ivo Georgiev (GEORGIEV, 10/2018) também foi executada no kit para verificar se a minha transformação do código em MATLAB para C foi feita corretamente e validar os resultados obtidos. Os resultados foram comparados e foi concluído que os tempos estão corretos, a diferença não foi significativa.

Em contrapartida, pela tabela 6 a resolução do UKF acelerando somente o cholesky não teve um resultado tão expressivo, cerca de 1,26 vezes mais rápido. O que representa uma redução de aproximadamente 20% no tempo de resolução, mas ainda é um valor significativa. Esse resultado era previsível pois somente o Cholesky foi acelerado, e conforme a tabela 1 e 5 o cholesky representa 20,8% do total, então ele foi minimizado mas o resto ainda representa uma parte expressiva do tempo total do cálculo.

Para melhorar esse resultado seria interessante adicionar mais funções ao co-processador. Mas é importante ressaltar que as funções do UKF implementada em C que gerou o resultado referente a tabela 1 não está otimizado, apenas o cholesky foi revisado e melhorado, conforme discutido em 4.1. Logo, o tempo total pode ser reduzido se o algoritmo for otimizado e assim o cholesky representaria uma parcela maior do tempo total de resolução, potencialmente equivaleria a 33% conforme a figura 3.

Sobre o clock do barramento, foi verificado que apesar do barramento suportar até 200MHz, não é possível utilizar um clock maior que 150MHz se a DMA for utilizada. Já sobre ao clock do co-processador, é possível fornecer até 217MHz.

Em relação as melhorias comentadas em 4.4.3 sobre alterar o IP do somador/subtrator para um com menos estágio e maior frequência, do ponto de vista do cálculo do UKF não é necessário, pois a melhoria de desempenho final seria mínima. Mas caso o co-processador seja utilizado para outros algoritmos que precisem do cholesky, pode ser interessante essa modificação pois diminuiria a quantidade de ciclos e permitiria fornecer um clock maior para o co-processador. Se essa melhoria for realizada o clock máximo passa a ser 300MHz.

5 Conclusão

O objetivo de acelerar o algoritmo UKF foi realizado com uma aceleração final de aproximadamente 25%, o que representa uma redução de 20% no tempo. Esse resultado pode ser suficiente para ser utilizado em um sistema crítico de tempo real. Caso haja necessidade de acelerar ainda mais, seria necessário adicionar mais funções ao co-processador, o que pode ser feito em um futuro trabalho.

O processador projetado possui características interessantes como:

- Acelera em até 26 % o algoritmo de UKF.
- Acelera em até 2006% a decomposição de cholesky.
- Permite a atribuição de dois clocks diferentes, assim a entrega e saída de dados teria o clock do barramento mas o núcleo pode ter um clock maior. Ou seja, se for otimizado os blocos de floating points é possível que ele utilize um clock mais rápido e aumente sua performance.
- O processador principal fica disponível para executar outras tarefas durante o tempo em que o co-processador está em execução, assim o tempo ocioso pode ser utilizado para tratar interrupções ou outros processos envolvidos. Como geralmente sistemas de tempo real utilizam sistemas operacionais é ainda mais proveitoso utilizar o co-processador visto que o sistema operacional para as atividades o tempo todo para lidar com outros
- Começa a processar os dados assim que eles começam a entrar, e escreve os dados à medida que forem calculados. Isso permite que o tempo morto de enviar dados e ler os dados seja minimizado.
- A desvantagem é o maior tempo de desenvolvimento.

Além disso, com a tendência de SoC e processadores especializados, o chip Cyclone-V-SoC tem potencial para se tornar muito utilizado e importante tanto na academia quanto na indústria, pois permite a prototipagem de SoCs e também pode ser utilizado para testes de IPs. Logo, a análise sobre o chip, seus recursos e sistema feito no capitulo 4.2.2 é muito útil para futuros trabalhos, visto que pode ser reaproveitada e adaptada, como por exemplo para um acelerador de rede neural.

Um ponto negativo do chip Cyclone V SoC é que devido a sua arquitetura incomum, existem poucos trabalhos publicados, mas é possível encontrar mais trabalhos com

seu concorrente Xilinx Zynq SoC. Outro ponto negativo é que possui uma forma de gravação nada intuitiva e não consegui depurar o software quando utilizado o HPS e FPGA simultâneos, apenas foi possível a depuração quando somente a HPS foi utilizada e com um software ARM-DS que é pago.

Concluo então que o objetivo do trabalho foi cumprido, sendo muito produtivo tanto como aplicação dos conhecimentos desenvolvidos ao longo do curso quanto como precedente para futuros trabalhos.

Referências

ASANO, T. M. S.; YAMAGUCH, Y. Performance comparison of fpga, gpu and cpu in image processing. *19th Int. Conf. Field Programmable Logic Appl.*, 2009. Citado na página 25.

BURNS, A. W. A. *Real-Time Systems and Programming Languages*. 4th. ed. [S.l.]: Addison-Wesley, 2009. Citado 2 vezes nas páginas 19 e 20.

CAO, Y. *An implementation of Unscented Kalman Filter for nonlinear state estimation.* 2008. Disponível em: <https://www.mathworks.com/matlabcentral/fileexchange/18217-learning-the-unscented-kalman-filter>. Citado na página 33.

GEORGIEV, I. *UKF Lib*. 10/2018. Disponível em: <https://github.com/ivo-georgiev/ukfLib>. Citado na página 60.

GUSTAFSSON, G. H. F. Some relations between extended and unscented kalman filters. *IEEE Transactions on Signal Processing*, v. 60, n. 2, 2012. Citado 3 vezes nas páginas 20, 26 e 27.

INTEL. *Cyclone V Device Datasheet*. [S.l.], 05/2018. CV-51002. Citado na página 36.

INTEL. *Intel - SoC Hardware Overview*. 11/2018. Disponível em: <https://www.intel.com/content/www/us/en/programmable/support/training/course/oemb5501.html>. Citado 2 vezes nas páginas 13 e 30.

INTEL. *Cyclone V Hard Processor System Technical Reference Manual*. [S.l.], 1/2018. $Cv_5v4. Citado na página 36.$

INTEL. *Floating-Point IP Cores User Guide*. [S.l.], 12/2016. UG-01058. Citado na página 55.

INTEL. *Cyclone V and Arria V SoC Device Design Guidelines*. [S.l.], 6/2018. AN 796. Citado 5 vezes nas páginas 13, 35, 36, 37 e 53.

JULIER, J. K. S. J. A new extension of the kalman filter to nonlinear systems. *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls. Signal Processing, Sensor Fusion, and Target Recognition VI*, 1997. Citado na página 26.

KESTUR, J. D. S.; WILLIAMS, O. Pblas comparison on fpga, cpu and gpu. *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2010. Citado na página 25.

MOLANES, J. J. R.-A. R. F. Performance characterization and design guidelines for efficient processor–fpga communication in cyclone v fpsocs. *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS*, v. 65, n. 5, 2018. Citado 4 vezes nas páginas 13, 25, 41 e 42.

PATTERSON, J. H. D. *Computer Organization and Design*. [S.l.]: Morgan Kaufmann. Citado na página 45.

PUTTY. *Putty*. 11/2018. Disponível em: <https://www.putty.org/>. Citado na página 54.

PX4-AUTOPILOT. 11/2018. Disponível em: <https://docs.px4.io/>. Citado na página 33.

RODRıGUEZ-ANDINA, M. D. V. J. J.; MOURE, M. J. Advanced features and industrial applications of fpgas—a review. *IEEE Trans. Ind. Inform.*, v. 11, n. 5, 2015. Citado na página 26.

TERASIC. *DE0-Nano-SoC Kit/Atlas-SoC Kit*. 11/2018. Disponível em: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=163&No=941&PartNo=3>. Citado 3 vezes nas páginas 13, 52 e 53.

WIKIPEDIA, T. F. E. *Embedded System*. 06/2018. Disponível em: <https://en.wikipedia.org/wiki/Embedded_system>. Citado na página 19.