Um Sistema Heterogêneo Embarcado para Aceleração de Interseção Raio-Triângulo

Adrianno A. Sampaio[1], **Alexandre C. Sena**[1], **Alexandre S. Nery**[2]

[1] Instituto de Matemática e Estatística (IME)
Universidade do Estado do Rio de Janeiro (UERJ)
Rio de Janeiro, RJ – Brasil

[2]Departamento de Engenharia Elétrica (FT/ENE)
Universidade de Brasília (UnB)
Brasília, DF – Brasil

{adriannosampaio,asena}@ime.uerj.br, anery@redes.unb.br

Abstract. *Image rendering is an important research area in computer graphics and is applicable to many situations such as games, architectural visualization, cinema, among others. Nowadays, rendering of realistic images is one of the main challenges, especially for real time applications, where the challenge lies in balancing between image fidelity and computational performance. The Ray-Tracing method has been one of the main algorithms used for realistic imaging due to its accurate modeling of optical phenomena, but its disadvantage is its high computational cost. Several algorithms and hardware platforms have been proposed and used so far to improve the performance of this algorithm, but solutions dependent on many-core architectures have a high power consumption despite the performance boost. Thus, the aim of this paper is to propose a heterogeneous CPU-FPGA system on a low-energy and low-cost platform, and to analyze its performance, scalability and load balancing between computational resources in rendering different image sizes.*

Resumo. *A renderização de imagens é uma importante área da computação gráfica, sendo aplicável a diversas áreas como jogos, visualização arquitetônica, cinema, entre outras. Atualmente a renderização de imagens realistas é um dos principais desafios, especialmente para aplicações em tempo real, sendo a maior dificuldade balancear entre realismo e desempenho computacional. O método de Ray-Tracing tem sido um dos principais algoritmos utilizados para a geração de imagens realistas por sua naturalidade ao modelar fenômenos ópticos com precisão, porém sua desvantagem é o seu alto custo computacional. Diversos algoritmos e plataformas de hardware têm sido utilizados até o momento para melhorar o desempenho deste algoritmo, porém soluções com arquiteturas baseadas em Many-core ou GPUs possuem um alto consumo energético apesar do desempenho obtido. Assim, o objetivo deste trabalho é propor um sistema heterogêneo CPU-FPGA em uma placa embarcada de baixo custo energético, e analisar seu ganho de desempenho, escalabilidade e balanceamento de carga entre recursos computacionais renderizando diferentes tamanhos de imagens.*

1. Introdução

Uma vez que a Lei de Moore atingiu o seu limite é cada vez maior a necessidade de se criar e usar sistemas e arquiteturas heterogêneas, em especial para possibilitar a extração de desempenho que muitas aplicações modernas necessitam. Concomitantemente a isto, os recursos do sistema heterogêneo, sejam eles computacionais ou de armazenamento, entre outros, precisam ser usados de forma eficiente, a fim de garantir sua aplicabilidade em situações em que a sustentabilidade e o baixo consumo energético são requisitos cada vez mais importantes.

A renderização de imagens é um relevante tópico de pesquisa na computação gráfica, especialmente para indústria de cinema e jogos. Atualmente, com o crescimento do poder computacional de GPUs (*Graphics Processing Units*) e processadores modernos, muitas máquinas são capazes de renderizar cenas relativamente complexas, inclusive em resolução 4K. Por outro lado, o custo energético destes dispositivos tem aumentado proporcionalmente, o que pode tornar proibitivo manter tais sistemas em execução, principalmente considerando-se que a renderização de cenas mais complexas ou de animações pode levar várias horas ou dias de trabalho constante por parte do hardware.

Neste contexto, este trabalho apresenta a implementação e avaliação de um sistema heterogêneo para aceleração eficiente do algoritmo de traçado de raios, em particular das suas operações de Interseção Raio-Triângulo. Tal sistema é voltado para arquiteturas heterogêneas e embarcadas, normalmente compostas de processadores ARM e chips reconfiguráveis (FPGA - *Field Programmable Gate Arrays*), tendendo a um consumo de energia menor que aquele normalmente apresentado por arquiteturas heterogêneas CPU/GPU. Mais importante, os experimentos mostram que a solução heterogênea ARM/FPGA é capaz de aproveitar o potencial de desempenho do sistema e a executar eficientemente o algoritmo de traçado de raios (*Ray-Tracing*).

O hardware utilizado para implementação do sistema foi a placa Digilent PYNQ-Z1, que possui um processador ARM Dual-Core Cortex-A9 650MHz, 512 MB RAM e uma FPGA XC7Z020-1CLG400C da família Zynq-7000 [Digilent 2017]. O sistema executa sob um sistema operacional baseado em linux (petalinux) e possui bibliotecas Python proprietárias da Xilinx para realizar a interface entre os processadores ARM e a FPGA. Foram implementadas e avaliadas 3 opções do sistema, aumentando progressivamente o uso dos recursos de hardware: Versão CPU (*Single-Core* e *Multi-Core*); Versão FPGA (com uma e duas instâncias do co-processador RT); e Versão Heterogênea (CPU x2 + FPGA x2). Todas estas implementações foram desenvolvidas utilizando o algoritmo de Möller-Tumbore [Möller and Trumbore 2005] para o cálculo de Interseção Raio-Triângulo, sendo diferentes apenas no tipo de hardware e no balanceamento de carga de trabalho. O código completo desenvolvido para este artigo pode ser encontrado em um repositório no *website github.com* [Sampaio 2019].

O trabalho está dividido da seguinte maneira: Trabalhos relacionados são descritos na Seção 2. A Seção 3 descreve a técnica de *Ray-Tracing*. Em seguida, a Seção 4 apresenta a proposta de implementação para as novas arquiteturas baseadas em ARM e FPGA. A avaliação de desempenho do algoritmo proposto é apresentada na Seção 5. Por fim, conclusões e trabalhos futuros são apresentados na Seção 6.

2. Trabalhos Relacionados

Esta seção expõe os trabalhos relacionados encontrados na literatura. Foram buscados trabalhos com foco em apresentar implementações de co-processadores em FPGA para *Ray-Tracer* [Malcheva and Yunis 2014, Todman and Luk 2001, Woop et al. 2005]. Porém, a maior parte dos trabalhos encontrados apresentam uma implementação em linguagens de definição de *hardware* e neles, todo o algoritmo de *Ray-Tracing* está implementado em hardware, incluindo a fase de sombreamento e cálculo de cores. Em contraste, nosso trabalho possui como foco apenas a etapa de cálculos de interseção raio-triângulo. Isto se dá porque, além de ser o trecho mais computacionalmente custoso do algoritmo, os cálculos de interseção são independentes de qualquer modelo de iluminação, podendo ser utilizados por qualquer aplicação que implemente o algoritmo, incluindo aplicações cujo foco não é renderização.

Park et al. [Park et al. 2008] apresentaram uma implementação do algoritmo em FPGA como um protótipo para desenvolvimento de um ASIC (*Application-Specific Integrated Circuit*) especializado. Neste trabalho, a cena tridimensional é construída em uma CPU e então enviada para ser renderizada na FPGA. Nery et al. [Nery et al. 2010] implementaram um *Ray-Tracer* em FPGA contendo seu próprio conjunto de instruções. Esta implementação também possui a função de gerar os raios primários além do algoritmo. Schmittler et al. [Schmittler et al. 2004] criaram uma implementação em tempo real de um *Ray-Tracer* em FPGA com suporte a todas as funções básicas necessárias no algoritmo. Collinson e Sinnen [Collinson and Sinnen 2017] apresentam uma proposta de arquitetura *cache* para *Ray-Tracer* com o objetivo de amenizar o custo de acesso constante à memória utilizando a política LRU. Deng et al. [Deng et al. 2017] apresentam uma pesquisa sobre técnicas para aceleração do algoritmo de *Ray-Tracing* desenvolvidas ao longo do tempo.

3. *Ray-Tracing*

O algoritmo de *Ray-Tracing* tem como objetivo traçar a trajetória dos raios de luz em uma cena tridimensional. Ele consiste, basicamente, em disparar raios de luz a partir de cada pixel de uma câmera virtual em direção a parte da cena que se deseja renderizar. Estes são conhecidos por raios primários. Isto é feito a fim de descobrir, para cada raio primário, o seu ponto de interseção com o objeto mais próximo de sua origem, caso exista um. O ponto de interseção e as características do objeto, então, são utilizados para calcular a cor final do pixel em questão. Existem também casos em que as propriedades de um dado objeto exigem a geração de novos raios (chamados de raios secundários), que são utilizados recursivamente para modelar efeitos ópticos como reflexão e refração. Uma implementação básica de um *Ray-Tracer* normalmente utiliza-se de 4 abstrações computacionais:

1. **Câmera Virtual:** responsável por armazenar informações sobre a imagem a ser renderizada, e informações necessárias para a geração dos raios primários;
2. **Objetos 3-D:** responsáveis por descrever as informações geométricas de uma cena virtual, como posição, forma e tamanho de um objeto;
3. **Materiais:** esta classe possui informações sobre as propriedades físicas da superfície/volume de um objeto 3D. Os materiais são responsáveis por descrever o comportamento da luz ao atingir uma superfície;

4. **Fontes de Luz:** descrevem pontos ou superfícies das quais a luz é emitida. São essenciais ao cálculo da cor de um objeto atingido por um raio.

A Figura 1 demonstra o comportamento de um raio virtual durante a execução do algoritmo e suas etapas. (1): Inicialmente um raio primário R_p é lançado a partir de um pixel na Câmera Virtual. (2): São realizados os cálculos de interseção e o raio R_p atinge um objeto esférico S_1. De acordo com as propriedades físicas de S_1 um novo raio secundário R_s é gerado e atinge um novo objeto S_2. (3): Após estas ações se repetirem, o raio R_s gera uma cor, representando a luz incidente em S_1. O somatório da cor retornada por R_s e a influência de cada fonte de luz no ponto de interseção, gera uma nova cor, retornada por R_p e é, então, armazenada no pixel correspondente.

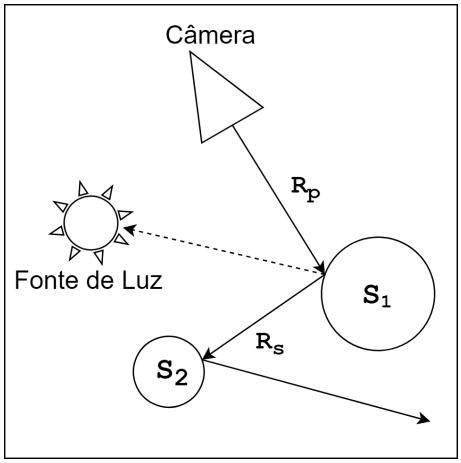


Figura 1. Visão geral do algoritmo de *Ray-Tracing*

A etapa (2), que realiza o *Cálculo de Interseções*, em geral é a parte mais computacionalmente custosa de um *Ray-Tracer* [Whitted 1980], o que torna esta etapa a principal candidata à tentativas de otimização e aceleração. Em sua versão mais simples, é necessário testar a interseção entre todos os raios e todos os objetos da cena tridimensional. Porém, uma das grandes vantagens do algoritmo é a sua natureza paralela. É possível escalar o desempenho de um *Ray-Tracer* de forma praticamente linear ao executar o cálculo de interseção de múltiplos raios em paralelo [Hurley 2005].

4. O Sistema Heterogêneo Embarcado

Esta seção descreve a arquitetura do sistema heterogêneo, suas particularidades e sua implementação usando ARM e lógica reprogramável FPGA.

4.1. A Arquitetura do Sistema

O sistema implementado consiste em um programa feito na linguagem Python, responsável pelo gerenciamento dos recursos de hardware e pela divisão da carga de trabalho entre a CPU e a FPGA. Os recursos utilizados e a divisão de trabalho são definidos pelo usuário estaticamente no começo da execução. Ao receber uma nova tarefa, o programa Python divide os dados de entrada entre os recursos computacionais, inicializando os cálculos de interseção em cada um deles. O sistema foi desenvolvido utilizando a linguagem Python para permitir o uso das bibliotecas do pacote *pynq* [Xilinx] além de tornar mais simples a divisão das tarefas entre a CPU e a FPGA.

O cálculo das interseções pode ser feito de 3 formas principais: apenas na CPU; apenas no acelerador em FPGA; de forma heterogênea (CPU + FPGA). Para isso, o sistema implementa duas classes específicas, que gerenciam os *hardwares* disponíveis e servem como uma interface entre o programa Python e estes recursos: *TracerCPU* e *TracerFPGA*. Em ambas as classes, está implementado um método `compute()` que inicia a execução dos cálculos de interseção no *hardware* correspondente.

A classe *TracerCPU* é responsável por executar os cálculos no processador ARM de forma sequencial ou paralela. Por questões de desempenho, seu funcionamento interno apenas recebe os dados de entrada da tarefa corrente e os executa em funções externas implementadas em C++. Estas funções são acessadas pelo programa Python através de *bindings* gerados com o uso da biblioteca *pybind11* [Jakob 2016]. A linguagem C++ também permite o uso da biblioteca *OpenMP* para a implementação *Multicore*, retirando esta responsabilidade da linguagem Python.

A segunda classe, chamada *TracerFPGA*, tem como função gerenciar o uso dos co-processadores implementados na FPGA. Esta classe armazena referências à posição de memória dos sinais de controle para cada instância do co-processador FPGA e aloca as estruturas de dados necessárias à sua execução. Internamente, a classe se utiliza do pacote *pynq* para enviar os sinais de controle necessários à comunicação com os co-processadores e para ordenar os dados corretamente na memória de forma que a FPGA consiga acessá-los. A implementação atual em FPGA possui duas instâncias do co-processador implementado para cálculos de interseção, que podem realizar tarefas paralelamente.

A Figura 2 mostra a diferença do comportamento interno das duas classes. Durante a execução do método `compute` da classe *TracerCPU*, é chamada uma das funções do módulo implementado em C++, podendo ser a versão sequencial ou paralela. Por outro lado, a classe *TracerFPGA* utiliza a biblioteca PYNQ para enviar os sinais de controle e dados de entrada ao co-processador. Ao contrário da versão CPU, o método `compute` da versão FPGA não bloqueia o fluxo de execução do programa, mas apenas sinaliza que o co-processador pode iniciar o processamento. Assim, é necessário verificar se a FPGA finalizou a execução e, após isso, retirar os resultados gerados com o método `get_results`.

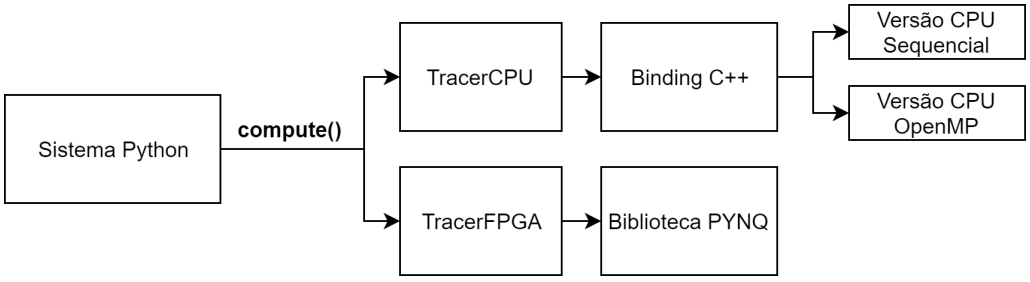


Figura 2. Estrutura geral do sistema Python.

4.2. Implementação do Sistema Heterogêneo

A renderização no sistema tem seu ponto de entrada em uma classe chamada *Renderer*, responsável principalmente por receber a tarefa a ser realizada, inicializar as estruturas de dados e dividir estaticamente a carga de trabalho entre os recursos que serão utilizados, no caso, o processador ARM Cortex-A9 e a FPGA XC7Z020-1CLG400C.

O construtor da classe *Renderer* possui a inicialização das classes *TracerCPU* e *TracerFPGA*, além de outras tarefas menores como leitura de arquivos de configuração. Ao inicializar a classe *TracerFPGA*, também é realizada a programação da FPGA com o *bitstream* obtido durante o processo de síntese. Em seguida, é executada a função *run()* desta classe, responsável por receber a tarefa a ser realizada, dividi-la seguindo a proporção especificada no arquivo de configuração (em uma variável chamada `fpga_load`), e iniciar a execução simultânea das sub-tarefas nas diferentes plataformas utilizadas.

Como cada raio precisa testar sua interseção com todos os triângulos, apenas os raios podem ser divididos entre diferentes recursos. A Figura 3 mostra que os triângulos de uma dada tarefa de renderização não são divididos, mas sim enviados em sua totalidade para cada núcleo ou instância. Em contraste, os raios são divididos em blocos menores, que são processados por cada recurso. Em casos nos quais apenas a CPU ou FPGA são utilizadas, a carga de trabalho dos raios é dividida igualmente entre os núcleos ou instâncias. Na implementação heterogênea, por outro lado, a FPGA é responsável por 40% do trabalho de processamento. Este balanceamento foi obtido com base em testes individuais de cada recurso e testes de balanceamento estático, e é o valor que apresenta o melhor desempenho na implementação atual.

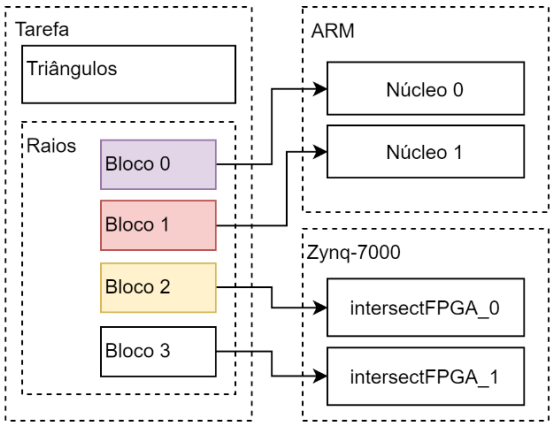


Figura 3. Distribuição de raios entre os diferentes recursos computacionais em uma tarefa.

4.3. Classe TracerCPU

Devido às diferenças de desempenho entre a linguagem Python e C++, a segunda foi escolhida para implementar os cálculos de interseção no processador ARM, principalmente por ser um trecho com um grande volume de cálculos matemáticos. A Listagem 1 mostra a implementação paralela da função de cálculo de interseções para o processador ARM em C++. A função é composta por dois *loops* aninhados iterando respectivamente por todos os raios e todos os triângulos, obtendo ao fim, o triangulo mais próximo atingido por cada raio. A linha 17 contém a chamada para a função que implementa o algoritmo de Möller-Trumbore. Como cada raio tem comportamento independente, o *loop* dos raios foi o escolhido para paralelização com a biblioteca *OpenMP*. Por fim, a função retorna uma lista dos triângulos atingidos por cada raio (ou -1 caso nenhum triângulo tenha sido atingido) e a distância percorrida pelo raio até atingir este triângulo.

```
1 intersectResults computeIntersectionsParallel(
2     std::vector<double> rayData,
3     std::vector<int> triangleIds,
4     std::vector<double> triangleData)
5 {
6     // Task information
7     int numTriangles = triangleData.size() / TRIANGLE_ATTR_NUMBER ;
8     int numRays = rayData.size() / RAY_ATTR_NUMBER;
9     // Output vectors
10    std::vector<int> outIds(numRays);
11    std::vector<double> outInter(numRays);
12    #pragma omp parallel for
13    for(int ray = 0; ray < numRays; ray++){
14        outIds[ray] = -1; outInter[ray] = INFINITY;
15        for(int tri = 0; tri < numTriangles; tri++){
16            double t;
17            if(rayIntersect(t, ray, rayData, tri, triangleData))
18            if(t < outInter[ray] && t > EPSILON){
19                outIds[ray] = triangleIds[tri];
20                outInter[ray] = t;
21            }
22        }
23    }
24    return std::make_pair(outIds, outInter);
25 }
```

Listagem 1. Código responsável pelo cálculo *multicore* de interseções Raio-Triângulo no processador ARM.

```
1 PYBIND11_MODULE(tracer, m) {
2     <...>
3     m.def("compute", &computeIntersections, "Compute Ray-Triangle intersections");
4     m.def("compute_parallel", &computeIntersectionsParallel, "Compute Ray-Triangle
          intersections with the parallel option");
5 }
```

Listagem 2. Código utilizado para expor as funções em C++ para cesso na linguagem Python.

É importante ressaltar que esta função implementa apenas a versão paralela do código, e a sua única diferença em relação à versão sequencial é a diretiva *pragma* contida na linha 12, que não é utilizada na versão sequencial. Este código foi utilizado em conjunto com a biblioteca *pybind11* para a criação do *binding* para a linguagem Python. O código utilizado para a criação do *binding* está na Listagem 2, onde as duas versões são incluídas no módulo Python.

4.4. Classe TracerFPGA

A implementação do co-processador em FPGA foi feita utilizando a linguagem C em conjunto com a ferramenta *Vivado HLS* [Xilinx 2017]. Esta ferramenta contém um compilador responsável por converter o código em uma linguagem de alto nível (no caso, C/C++) para uma linguagem de definição de *hardware* (HDL), como VHDL ou Verilog. O código criado possui uma implementação similar à implementação em CPU, porém com algumas adaptações e diretivas exigidas pelo compilador HLS.

A Listagem 3 mostra a estrutura do código HLS que implementa o co-processador. É possível notar que os parâmetros são diferentes da versão implementada para a CPU apenas no fato de que estão sendo passados como ponteiros para *arrays* de C e existem valores inteiros representando os seus tamanhos. Os parâmetros são:

- **i_tNumber:** Número de triângulos enviados ao co-processador;
- **i_tData:** Informações geométricas dos triângulos da cena;
- **i_tIds:** Identificadores dos triângulos enviados ao co-processador;
- **i_rNumber:** Número de raios que serão processados pelo co-processador;
- **i_rData:** Informações geométricas dos raios;
- **o_tIds:** Identificador do triângulo interceptado mais próximo de cada raio (-1 caso não exista interseção);
- **o_tIntersects:** Distância entre o triângulo mais próximo de um raio e a origem deste raio;

Além disso, estão sendo utilizados apenas dois tipos de diretivas de interface do compilador HLS: *s_axilite*, que descreve a transferência de dados com o protocolo AXI-Lite; e, *m_axi*, que descreve o uso do protocolo de comunicação AXI-Full.

O protocolo AXI-Lite é um protocolo que permite a transferência de apenas um dado por vez, sendo necessário testar se o barramento não está em uso antes da próxima transferência. Este fato torna o protocolo realmente útil para transferência de sinais de controle, porém não é eficiente para transferência de maiores volumes de dados. Por outro lado, o protocolo AXI-Full (uma generalização do protocolo AXI-Lite) permite a transferência de dados em modo de rajada, permitindo que mais dados sejam lidos com uma única requisição. Por isto, os inteiros contendo o tamanho dos *arrays* e os endereços de memória destes *arrays* são enviados ao acelerador pelo protocolo AXI-Lite, enquanto a leitura dos seus dados é feita através do protocolo AXI-Full.

```
1 void intersectFPGA(int i_tNumber, volatile double *i_tData, volatile int *i_tIds,
2     int i_rNumber, volatile double *i_rData, volatile int *o_tIds, volatile double *o_tIntersects)
3 {
4     //interface definition pragmas
5     #pragma HLS INTERFACE s_axilite port=return bundle=control
6     #pragma HLS INTERFACE s_axilite port=i_tData bundle=control
7     #pragma HLS INTERFACE m_axi depth=450000 port=i_tData
8     #pragma HLS INTERFACE s_axilite port=i_tIds bundle=control
9     #pragma HLS INTERFACE m_axi depth=50000 port=i_tIds
10    #pragma HLS INTERFACE s_axilite port=i_rData bundle=control
11    #pragma HLS INTERFACE m_axi depth=12441600 port=i_rData
12    #pragma HLS INTERFACE s_axilite port=o_tIds bundle=control
13    #pragma HLS INTERFACE m_axi depth=2073600 port=o_tIds
14    #pragma HLS INTERFACE s_axilite port=o_tIntersects bundle=control
15    #pragma HLS INTERFACE m_axi depth=2073600 port=o_tIntersects
16    #pragma HLS INTERFACE s_axilite port=i_tNumber bundle=control
17    #pragma HLS INTERFACE s_axilite port=i_rNumber bundle=control
18    //main ray-processing loop
19    for(ray = 0; ray < nRays; ray++){
20        for(tri = 0; tri < i_tNumber; tri++){
21            <...> // Moller-Trumbore algorithm
22        } // triangle loop end
23    } // rayLoop
24 }
```

Listagem 3. Código Vivado HLS do acelerador implementado.

Este código HLS, após sua compilação, gera um IP (Propriedade Intelectual) com a definição do acelerador em uma linguagem de definição de hardware. Este IP é importado para um projeto no *software* Vivado, que realiza a síntese do acelerador na FPGA. Durante a criação do projeto é também possível replicar o co-processador em diferente instâncias. Na placa utilizada para este trabalho só foi possível incluir duas instâncias do

co-processador. Na Figura 4, (1) e (2) sinalizam ambas as instâncias do co-processador, e (3) sinaliza o IP de iterface com o restante dos recursos da placa PYNQ-Z1, como a memória RAM, interfaces I/O e o processador ARM.

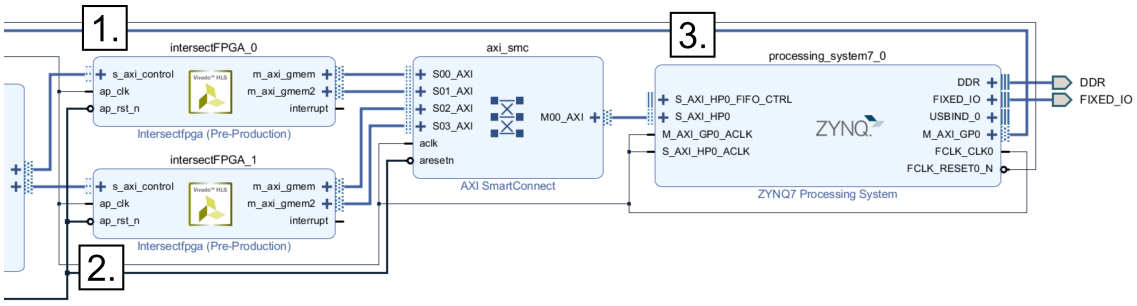


Figura 4. Diagrama de blocos do Vivado

5. Resultados Experimentais

Os experimentos foram realizados apenas utilizando recursos da placa PYNQ-Z1, e as métricas utilizadas para avaliar o sistema são os tempos de execução da renderização, o *speed-up* das implementações em relação à versão CPU sequencial (rodando no ARM 650MHz) e o custo de área e recursos da FPGA. Os casos de teste utilizados são renderizações de imagens com resoluções de 100x100, 200x200, 300x300, 400x400 e a cena 3D é composta por 2000 triângulos. Estes casos são avaliados em 5 diferentes configurações:

1. ARM x1: *Single-Core*
2. ARM x2: *Multi-Core*;
3. FPGA x1: Uma instância do co-processador
4. FPGA x2: Duas instâncias do co-processador;
5. Versão Heterogênea (ARM x2 + FPGA x2)

5.1. Análise de Desempenho

Os experimentos realizados demonstraram que, com exceção da versão FPGA com uma instância, todas as outras versões obtiveram um desempenho melhor do que o obtido na versão ARM x1. A Tabela 1 mostra a média dos tempos de execução obtidos após 10 execuções do programa e a Tabela 2 mostra os coeficientes de variação dos tempos de execução para cada cenário de teste. Os baixos valores obtidos através do coeficiente de variação demonstram a precisão dos resultados. Como foi dito anteriormente, a versão heterogênea divide a tarefa passando 40% do trabalho para a FPGA. Esse balanceamento foi escolhido com base nos tempos obtidos nas versões ARM x2 e FPGA x2 com o objetivo de obter o máximo de desempenho disponível nesta configuração. Um gráfico com a visualização dos tempos de execução obtidos pode ser encontrado na Figura 5a.

Na Tabela 3 estão descritos os valores de *speedup* para cada uma das plataformas de *hardware* utilizadas. Pelos resultados obtidos a versão FPGA x1 foi a única que obteve desempenho inferior à versão ARM x1. Isto pode ocorrer devido a fatores como a latência de acesso à memória RAM da placa ou baixo paralelismo a nível de instrução. Também é possível notar que ao dobrar o número de núcleos (ARM) ou instâncias (FPGA) das implementações o ganho foi praticamente linear, confirmando a natureza embaraçosamente paralela do *Ray-Tracer*.

Tabela 1. Tempos de renderização das imagens por plataforma de *hardware*.

Tempo(s)	ARM x1	ARM x2	FPGA x1	FPGA x2	Heterogênea
100x100	25,25	13,26	37,11	18,84	8,24
200x200	100,51	51,64	148,11	75,07	31,48
300x300	226,50	116,00	333,05	168,73	70,59
400x400	402,72	206,46	591,86	299,56	125,39

Tabela 2. Coeficientes de variação dos tempos obtidos para cada cenário de teste.

CV(%)	ARM x1	ARM x2	FPGA x1	FPGA x2	Heterogênea
100x100	0,078%	0,464%	0,122%	0,054%	0,163%
200x200	0,184%	0,051%	0,002%	0,012%	0,040%
300x300	0,176%	0,039%	0,002%	0,014%	0,099%
400x400	0,163%	0,073%	0,006%	0,006%	0,061%

5.2. Análise de Uso de Recursos da FPGA

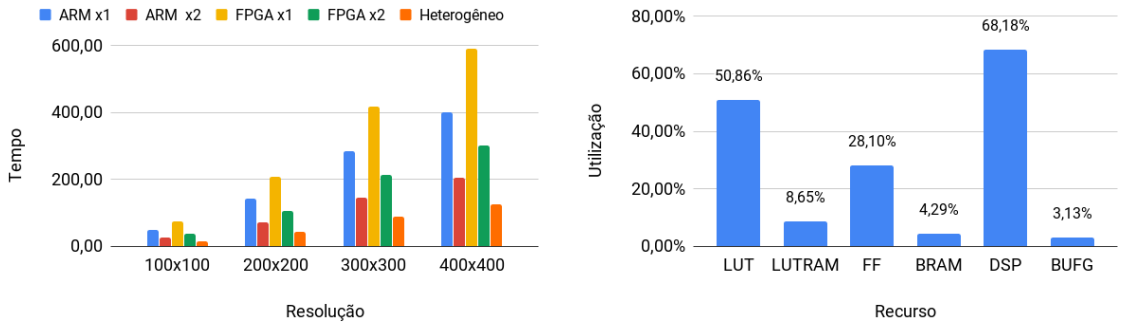
A utilização de recursos da FPGA composta de duas instâncias pode ser vista na Figura 5b, onde é possível observar que não é possível utilizar mais do que duas instâncias do acelerador concebido, pelo fato de que a placa PYNQ-Z1 não possui componentes DSP suficientes. Porém, a porcentagem de uso de recursos não foi alta no geral, o que deixa aberta a possibilidade de melhorias nas instâncias existentes.

Um dos principais pontos com possibilidade de melhoria observados através do gráfico de uso de recursos é o baixo uso das BRAMs. No gráfico (Figura 5b) é possível ver que o uso de BRAMs está em 4,29% para a versão com duas instâncias, ou seja, um uso baixo que poderia ser ampliado com um sistema de hierarquia de memória. As BRAMs são componentes de memória cujo acesso é mais rápido do que a memória RAM padrão da placa. Assim, com estes recursos disponíveis é possível aumentar o uso das BRAMs para fazer um uso mais eficiente da transferência de dados com o protocolo AXI-Full.

6. Conclusão e Trabalhos futuros

Por serem equipamentos de baixo custo energético, sistemas embarcados são de grande importância especialmente em aplicações que demandam grande poder computacional por longos intervalos de tempo. Para fazer uso desta vantagem, este trabalho apresenta um sistema heterogêneo de baixo custo energético e compara seu desempenho ao utilizar diferentes plataformas de *hardware* (CPU, FPGA ou ambos). Duas unidades de cálculos de interseção para *Ray-Tracer* foram desenvolvidas: a primeira foi desenvolvida para FPGA com o uso da ferramenta Vivado HLS, contando com até duas instâncias do co-processador desenvolvido; a segunda unidade foi desenvolvida em C++ para o processador ARM da placa contando com a possibilidade de execução *multicore* através da biblioteca OpenMP.

A partir dos dados obtidos também podemos afirmar que, ao aumentar o número de núcleos/intâncias utilizados pela placa PYNQ-Z1, obtemos um ganho de desempenho com comportamento aproximadamente linear devido à independência entre o processa-



(a) Gráfico de tempo de execução por plataforma de hardware

(b) Gráfico uso de recursos da FPGA pelo co-processador implementado.

Figura 5. Gráficos de resultados dos tempos de execução e uso de recursos da versão FPGAx2 do acelerador.

Tabela 3. Valores de *speedup* por plataforma de *hardware*.

ARM x1	ARM x2	FPGA x1	FPGA x2	Heterogeneous
-	1,939	0,680	1,341	3,170

mento de diferentes raios. Também deve ser levado em consideração que a frequência de operação da FPGA utilizada é de 100MHz, sendo inferior aos 650MHz do processador ARM. Ainda assim, na versão heterogênea, a FPGA x2 foi capaz de realizar 40% do trabalho total no mesmo intervalo de tempo em que o ARM x2 realizou os outros 60%.

Com isso, podemos concluir que a FPGA apresentou um bom desempenho em relação ao processador ARM contando ainda com recursos extras para a implementação de melhorias. Além disso, a versão heterogênea conseguiu aproveitar de forma eficiente todos os recursos computacionais do *hardware* utilizado, atingindo um *speedup* maior do que 3 vezes o desempenho da versão sequencial na CPU. Este projeto foi também realizado em uma placa de processamento embarcado e de baixo custo energético, o que o torna promissor como uma unidade de processamento individual, e também como um nó ser replicado em um sistema distribuído, *Cluster* ou *Grid*. Devido à falta de sensores na placa PYNQ-Z1 não foi possível obter o consumo de energia atingido, porém em trabalhos futuros estes números também serão avaliados.

Por fim, no futuro será implementada e avaliada uma versão distribuída do *Ray-Tracer* capaz de executar em uma única placa ou em um ambiente composto de várias placas de baixo consumo energético. Para isso, será desenvolvido um balanceamento de carga dinâmico capaz de distribuir eficientemente a carga entre os recursos.

Referências

Collinson, S. and Sinnen, O. (2017). Caching architecture for flexible fpga ray tracing platform. *Journal of Parallel and Distributed Computing*, 104:61–72.

Deng, Y., Ni, Y., Li, Z., Mu, S., and Zhang, W. (2017). Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Computing Surveys (CSUR)*, 50(4):58.

Digilent (2017). *PYNQ-Z1 Board Reference Manual*. Digilent Inc.

Hurley, J. (2005). Ray tracing goes mainstream. *Intel Technology Journal*, 9(2).

Jakob, W. (2016). pybind11 — seamless operability between c++11 and python. `https://github.com/pybind/pybind11`. Online, Acessado: 10-08-2019.

Malcheva, R. and Yunis, M. (2014). An acceleration of fpga-based ray tracer. *European Scientific Journal, ESJ*, 10(7).

Möller, T. and Trumbore, B. (2005). Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM.

Nery, A. S., Nedjah, N., and França, F. M. G. (2010). A parallel architecture for ray-tracing. In *2010 First IEEE Latin American Symposium on Circuits and Systems (LASCAS)*, pages 77–80.

Park, W. C., ho Nah, J., Park, J. S., Lee, K.-H., Kim, D.-S., Kim, S.-D., Park, J. H., Kim, C.-G., Kang, Y.-S., Yang, S.-B., and Han, T.-D. (2008). An fpga implementation of whitted-style ray tracing accelerator. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 187–187.

Sampaio, A. (2019). Heterogeneous ray-tracing pynq. `https://github.com/adrianno3259/heterogeneous-raytracing-pynq`. Online, Acessado: 22-09-2019.

Schmittler, J., Woop, S., Wagner, D., Paul, W. J., and Slusallek, P. (2004). Realtime ray tracing of dynamic scenes on an fpga chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pages 95–106, New York, NY, USA. ACM.

Todman, T. and Luk, W. (2001). Reconfigurable designs for ray tracing. In *null*, pages 300–301. IEEE.

Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349.

Woop, S., Schmittler, J., and Slusallek, P. (2005). Rpu: a programmable ray processing unit for realtime ray tracing. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 434–444. ACM.

Xilinx. Pynq - python productivity for zynq. `https://github.com/Xilinx/PYNQ`. Online, Acessado: 10-08-2019.

Xilinx (2017). *Vivado Design Suite User Guide: High Level Syntesis*. Xilinx.