Proposta e Implementação de um Acelerador Eficiente em HLS para *Ray Tracing*

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

[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

Resumo. *As arquiteturas reconfiguráveis, como os chips FPGA, apresentam um alto potencial computacional a um baixo custo energético. Assim, o objetivo deste trabalho é propor e implementar um acelerador em síntese de alto nível (HLS) para execução eficiente do algoritmo de Ray-Tracing (i.e técnica de grande importância na área de renderização de imagens, especialmente por seus resultados fisicamente precisos, apesar do seu alto custo computacional) nas FPGAs. Para isso várias estratégias de otimizações são propostas e avaliadas. Os resultados mostram que a versão mais rápida conseguiu alcançar resultados* 2 *vezes mais rápidos do que a versão para o processador ARM. Mais importante, os resultados obtidos ficaram bem próximos de dois* benchmarks *de limite inferior propostos o que mostra a eficiência dos aceleradores.*

1. Introdução

Ray-Tracing é considerado um dos principais algoritmos capazes de renderizar imagens realistas e fisicamente corretas [Glassner, 1989]. Porém, a síntese de imagens realistas tem um custo computacional muito alto. Por outro lado, para reduzir o tempo de execução é comum o uso de arquiteturas *multicore* (CPU) e *many-core* (GPU) para explorar a natureza embaraçosamente paralela do algoritmo [Hurley, 2005]. Esses sistemas, apesar de serem eficientes em relação ao tempo de execução, podem conseguir isso a um custo energético muito alto.

Uma alternativa a estes *hardwares* com demandas energéticas cada vez maiores são as arquiteturas reconfiguráveis, como FPGAs (*Field-Programmable Gate Arrays*), que têm ganhado espaço devido ao seu baixo custo energético [Wilson, 2011]. Estas plataformas apresentam a possibilidade de simular o comportamento de um *hardware* de propósito específico sem a necessidade de passar pelo processo de fabricação de um *chip ASIC*. Mais recentemente, o aumento do uso de FPGAs nas áreas de computação de alto desempenho e sistemas embarcados se deve principalmente à fabricação de FPGAs cada vez mais eficientes e com uma maior facilidade de uso proporcionada por compiladores de síntese de alto nível (HLS) [Xilinx, 2017, Chu, 2006].

Assim, este artigo propõe estratégias de otimização para execução eficiente do algoritmo de cálculo de interseção raio-triângulo em FPGAs. Nenhuma implementação fez

uso de estruturas de subdivisão espacial, para comparar o desempenho do algoritmo de interseção raio-triângulo. A partir destas estratégias propostas, diferentes versões foram implementadas utilizando o compilador HLS da ferramenta *Xilinx HLx Suite*. Além disso, para avaliar o desempenho das versões do acelerador, foram implementados dois *benchmarks* em FPGA e uma versão do algoritmo na linguagem C++ para servir como base de comparação. Enquanto que os *benchmarks* propostos funcionam como um limite inferior de tempo (ou limite superior de desempenho) para analisar quão perto/longe do valor ideal as otimizações propostas se encontram, a implementação em C++ é usada como base de comparação de desempenho, uma vez que as novas placas FPGA, na sua maioria, vem acopladas com processadores de propósito geral ARM [Digilent, 2017].

Os resultados mostraram que a versão proposta mais rápida apresentou fator de aceleração de 2x, enquanto que a melhor versão paralela obteve um fator de aceleração de 4,73x, ambos em comparação com a implementação sequencial do processador ARM da placa.

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

Nesta seção se encontram trabalhos similares nos quais o objetivo era o uso de FPGAs para a implementação eficiente, ou aceleração do algoritmo de *Ray-Tracing* [Malcheva and Yunis, 2014, Todman and Luk, 2001, Woop et al., 2005]. A maior parte dos trabalhos encontrados realizava a implementação em uma linguagem de definição de *hardware* (HDL), em geral VHDL, e nestes trabalhos o *Ray-Tracer* era completamente implementado em *hardware* (desde a geração de raios até o cálculo de materiais e iluminação.

Park et al. [2008] desenvolveram um protótipo de acelerador para *Ray-Tracer* em FPGA com o objetivo de avaliar a eficiência de uma implementação ASIC. Nery et al. [2010] implementaram um processador para *Ray-Tracer* em FPGA com um conjunto de instruções para o cálculo de pontos de interseção e a geração de raios em uma Unidade de Geração de Raios interna. Schmittler et al. [2004] criaram um protótipo de *Ray-Tracer* em tempo real capaz de uma performance entre 20 e 60 quadros por segundo. Outros *designs* [Fender and Rose, 2003] seguem a mesma ideia de implementação de um *Ray-Tracer* em FPGA. Collinson and Sinnen [2017] propuseram uma arquitetura de *cache* para aliviar os gargalos de memória em um *Ray-Tracer* que se utiliza da política de troca LRU. Além disso, Deng et al. [2017] realizaram uma pesquisas sobre técnicas de aceleração de *hardware* para aplicações do algoritmo de *Ray-Tracer*.

3. Algoritmo de Interseção Raio-Triângulo

O conceito básico do cálculo de interseção raio-triângulo é simples e pode ser definido pelo Algoritmo 1. O algoritmo recebe como entrada de dados 3 vetores: as coordenadas dos vértices de cada triângulo da cena 3D; os identificadores de cada triângulo; e, os dados de origem e direção de cada raio lançado à cena. Como valores de saída o algoritmo retorna dois vetores cujo tamanho é igual ao número de raios: o primeiro vetor contém

o identificador do triângulo interceptado por cada raio nos vetores de entrada (ou -1 caso não haja interseção); o segundo contém a distância percorrida pelo raio partindo de sua origem até o primeiro triângulo interceptado. Durante a execução do algoritmo, é feita uma iteração através dos raios e, para cada raio, é feito um cálculo de interseção entre este raio e todos os triângulos da cena obtendo-se, por fim, o triângulo mais próximo (caso este exista). Para o cálculo de interseção entre um raio e um triângulo é utilizado o, já conhecido, algoritmo de Möller-Trumbore [Möller and Trumbore, 2005].

```
Entrada: Triângulos, Identificadores de Triângulos, Raios
Saída: Distâncias, Triângulos Interceptados
para cada raio em Raios faça
    real menorDistância = 10^9;
    inteiro triânguloMaisPróximo = -1;
    para cada par (id, triângulo) em Identificadores e Triângulos faça
        booleano interseção;
        real distância;
        interseção, distância = moller_trumbore(raio, triângulo);
        se interseção == Verdadeiro E distância < menorDistância então
            menorDistância = distância;
            triânguloMaisPróximo = id;
        fim
    fim
    Distâncias[raio] = menorDistância;
    TriângulosInterceptados[raio] = triânguloMaisPróximo;
fim
```

Algoritmo 1: Algoritmo de cálculo de interseções de um *Ray-Tracer*.

Após iterar por todos os raios, o algoritmo está completo, sendo possível utilizar os resultados na próxima etapa do *Ray-Tracer*. Note que, as iterações dos raios são completamente independentes. Este fato é aproveitado em todas as versões implementadas através da criação de diferentes instâncias do acelerador durante a síntese do modelo, e dividindo a carga de trabalho entre estas instâncias.

4. Estratégias de Otimização Implementadas em HLS

A partir das características do cálculo de interseção raio-triângulo foram propostas 5 estratégias de otimização para a criação de aceleradores para FPGAs, sendo elas: (i) *Naive v1*; (ii) *Naive v2*; (iii) *BRAM*; (iv) *Pipeline*; (v) *Array Partition*. As subseções a seguir descrevem cada uma delas.

4.1. Versão Naive v1

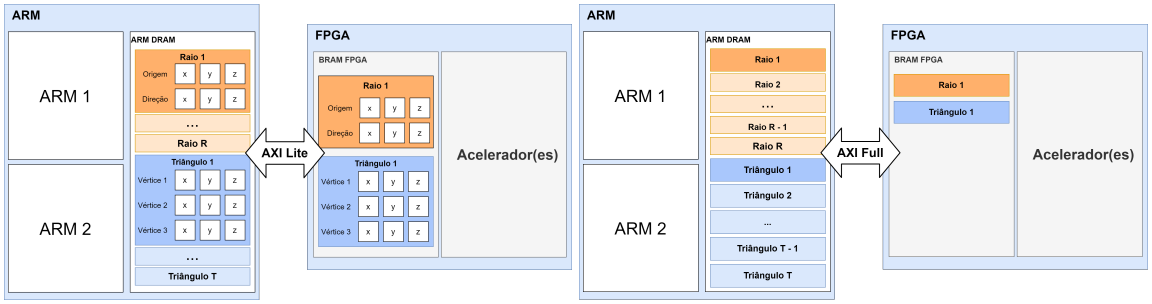
Esta subseção descreve o acelerador *Naive* v1. Duas implementações desta versão foram realizadas: versão com números reais de precisão dupla e precisão simples. Como esta é a única diferença entre as implementações *Naive* v1, os exemplos e listagens serão apresentados apenas com a versão em precisão simples. Uma análise da versão com números reais de precisão dupla foi apresentada em [Sampaio et al., 2019].

A versão inicial do acelerador para a FPGA foi criada a partir de uma implementação direta do algoritmo de Möller-Trumbore aplicada entre todos os raios e

todos os triângulos. Um diagrama da versão Naive v1 pode ser visto na Figura 1a, assim como a movimentação de dados entre a memória DRAM e as BRAMs da FPGA. Na figura, é possível observar a cópia dos dados de um raio e de um triângulo da memória RAM para as BRAMs. Essas duas memórias, existentes em todas as placas FPGA, são conectadas pela interface AXI-Lite. Assim, para cada componente x, y e z de cada vetor é necessário realizar toda a troca de mensagem do protocolo AXI-4 assim como a verificação de disponibilidade do barramento.

4.2. Versão Naive v2

Para reduzir o gargalo de comunicação, a segunda estratégia proposta utiliza o protocolo AXI-Full e sua cópia de dados em modo rajada, permitindo que a obtenção do endereço e verificação de disponibilidade do barramento sejam feitas uma única vez a cada raio ou triângulo, ao invés de ter que fazer isso para cada coordenada. A Figura 1b mostra o funcionamento da cópia de dados utilizando o barramento AXI-Full que permite a cópia no modo rajada. Assim, as coordenadas x, y e z dos vetores não são mais copiados separadamente. Porém esta versão mantém a cópia apenas do raio e do triângulo que estão sendo processados no momento.



(a) Versão *Naive v1*. **(b)** Versão *Naive v2*.

Figura 1. Diagrama de cópia de dados das versões *Naive v1* e *Naive v2*.

4.3. Versão BRAM

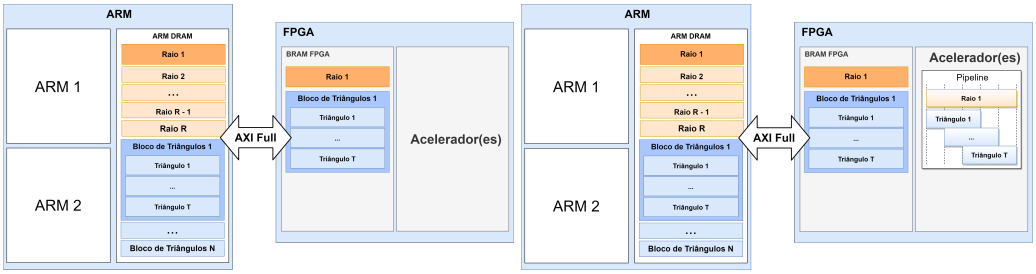
A versão chamada de *BRAM* pode ser vista como uma evolução natural da versão *Naive v2*, no sentido de que é uma otimização sobre a forma de como os dados são copiados da memória RAM para as BRAMs da FPGA.

O objetivo é aumentar a utilização das BRAMs, por serem muito velozes, necessitando apenas de um ciclo de *clock* para acessar um valor. Assim, a versão *BRAM* copia um bloco de triângulos por vez, ao invés de um único triângulo, através do barramento AXI-Full, como mostra a Figura 2a. Esta abordagem reduz o número de acessos à memória e verificações de disponibilidade de barramento durante a execução do algoritmo. Ao acabar o processamento dos triângulos do bloco, um novo bloco é copiado para a BRAM até que os blocos acabem, finalizando a iteração de um raio.

4.4. Versão Pipeline

A versão *Pipeline* tem como objetivo principal aproveitar melhor a velocidade de acesso aos dados armazenados nas BRAMs, aplicando a técnica de pipeline durante o processamento do bloco de triângulos. Como pode ser visto na Figura 2b, a movimentação de

dados não possui diferenças em relação à versão BRAM. A diferença, porém, se encontra na parte de processamento desses triângulos. Com o uso da técnica de pipeline, o processamento da interseção entre um raio R e um triângulo T é dividida em estágios que podem ser executados simultaneamente. Com isso, ao terminar o primeiro estágio do processamento de R e T, é possível iniciar o primeiro estágio de processamento entre R e $T + 1$, e assim por diante até ser necessário a cópia de um novo bloco.



(a) Versão *BRAM*. **(b)** Versão *Pipeline*.

Figura 2. Diagrama das versões *BRAM* e *Pipeline*.

4.5. Versão Partition

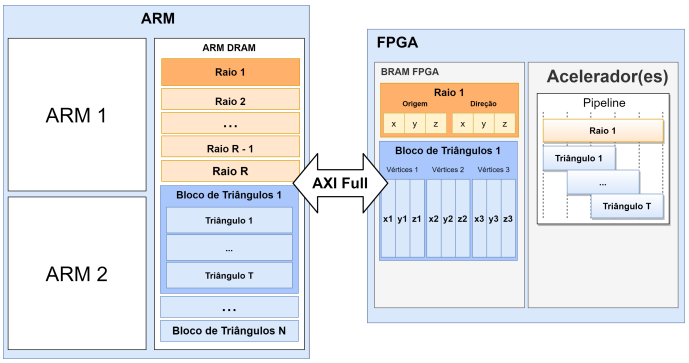
A característica adicionada nesta versão está relacionada à maneira como o armazenamento de dados nas BRAMs estão organizados. Uma BRAM é um componente de memória da FPGA cujo acesso é consideravelmente mais rápido do que um acesso à memória DRAM, podendo ser utilizada como uma forma simplificada de *cache* do acelerador. Porém na FPGA utilizada, as BRAMs possuem apenas duas portas de acesso, o que permite apenas 2 acessos simultâneos. Para superar esta limitação, o compilador HLS possui uma diretiva chamada `#pragma HLS ARRAY_PARTITION`, que permite que um *array* possa ser organizado nas BRAMs.

O reflexo da aplicação desta diretiva no conceito do acelerador pode ser observado na Figura 3 que mostra como os componentes de maior granularidade (raio e bloco de triângulos) são divididos ao serem copiados para as BRAMs da FPGA. No caso de um raio, onde existem apenas 6 componentes reais, cada componente é direcionado para uma BRAM diferente, de forma a permitir seu acesso paralelo. Como apenas um triângulo é acessado por vez, apenas este triângulo precisa ser completamente acessível. Isto é realizado dividindo-o em 9 diferentes blocos de forma cíclica, ou seja, os componentes x_1 de todos os triângulos são posicionados no mesmo bloco, os componentes y_1 serão posicionados em outro e assim por diante para todos os componentes. Desta forma, todos os componentes do triângulo estarão em BRAMs diferentes, permitindo um acesso paralelo.

4.6. Benchmarks

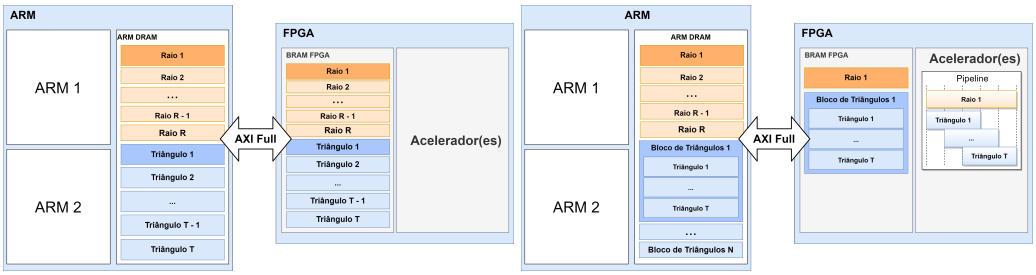
Para avaliar o desempenho das estratégias de otimização para o cálculo da interseção raio-triângulo propostas neste trabalho, foram desenvolvidos 2 *benchmarks* para servir como limite inferior para o tempo de execução chamados de (i) *BRAM Benchmark* e (ii) *Pipeline Benchmark*. A principal característica comum aos dois *benchmarks* é armazenar todos os dados a serem calculados na BRAM, evitando o custo de transferência de dados durante a execução.

Figura 3. Diagrama da divisão dos dados da versão *Partition*.



O primeiro *benchmark* consiste em uma implementação ideal do acelerador *BRAM* (Seção 4.3) na qual todos os raios e triângulos são copiados na sua totalidade para as BRAMs da FPGA através da interface de comunicação AXI-Full. Uma representação desse *benchmark* pode ser visto na Figura 4a, onde todos os dados de entrada necessários para os cálculos de interseção (raios e triângulos) são copiados de uma só vez do bloco *ARM DRAM* para o bloco *BRAM FPGA*.

A segunda versão de referência é o *Pipeline Benchmark* que, além de copiar todos os dados da cena para a memória BRAM como no *benchmark* anterior, também aplica o conceito de *Pipeline* apresentado na Subseção 4.4. Essa versão não só aproveita o fato de todos os dados necessários ao processamento do cálculo de interseção já estarem nas BRAMs, assim como aplica a técnica de pipeline durante o processamento do bloco de triângulos. Além disso, como não há um intervalo para a cópia de um novo bloco de triângulos, o processamento em *pipeline* não precisa ser interrompido. Esta configuração pode ser observada na Figura 4b.



(a) *BRAM Benchmark*. **(b)** *Pipeline Benchmark*.

Figura 4. Diagrama dos *benchmarks BRAM* e *Pipeline*.

5. Resultados Experimentais

Esta seção apresenta os resultados experimentais obtidos pelos aceleradores de *hardware* propostos e implementados neste trabalho. Estes resultados correspondem aos tempos de execução obtidos com as versões do acelerador e resultados de uso de recursos obtidos após a etapa de síntese de alto nível (*Vivado HLS*) e síntese HDL (*Vivado*). O ambiente de teste utilizado foi a placa PYNQ-Z1, da família Zynq-7000, contendo um processador ARM cortex-A9 dual-core, 512 MB de memória DRAM, e uma parte programável con-

tendo uma FPGA ZYNQ XC7Z020-1CLG400C. Todos os resultados apresentados equivalem a média aritmética de 30 execuções, onde o tempo está sempre em segundos. É importante destacar que os resultados obtidos são bastante confiáveis, onde o coeficiente de variação para todos os experimentos ficou sempre abaixo de 1%.

5.1. Resultados de Síntese

Ao criar o acelerador com a ferramenta HLS, é necessário instanciá-lo na placa FPGA, sendo possível que múltiplas instâncias sejam criadas e utilizadas. Essas instâncias são independentes entre si, podendo ser iniciadas completamente em paralelo, funcionando conceitualmente como um núcleo de processamento independente. Com isso, é possível aproveitar a natureza embaraçosamente paralela do algoritmo de *Ray-Tracer*, fazendo com que diferentes instâncias recebam diferentes blocos de raios a serem processados. A Tabela 1 apresenta o número de instâncias que puderam ser sintetizadas para cada versão do acelerador.

Tabela 1. Número máximo de aceleradores instanciados para cada versão.

	Naive v1		**Naive v2**	**BRAM**	**Pipeline**	**Partition**
	Float	**Double**				
Instâncias	6	2	6	6	5	4

Após sintetizada, a versão *Naive v1 (double)* pôde ser instanciada na FPGA duas vezes, enquanto na versão *float*, foi possível criar seis instâncias do acelerador, como também nas versões *Naive v2* e *BRAM*, abrindo a possibilidade para menores tempos de execução. As versões *Pipeline* e *Array Partition* permitiram a síntese de 5 e 4 instâncias do acelerador, respectivamente. Com isso podemos concluir que a versão com tipo de dados **double** limitou consideravelmente o desempenho do acelerador, suportando apenas duas instâncias na versão mais lenta.

Em relação aos resultados de uso de recursos da FPGA para cada uma das versões, apresentado na Figura 5, podemos ver que, com exceção da versão *Naive v1 (**double**)*, as *Look-Up Tables* (LUTs) e os DSPs foram bem aproveitados por todas as versões, sendo estes os fatores que limitaram a criação de mais instâncias. As BRAMs, por outro lado passaram a ter um uso maior a partir da versão *BRAM*, na qual é introduzida a cópia de triângulos da memória DRAM para as BRAMs em blocos. Também podemos notar que a queda do número de instâncias possíveis na versão *Array Partition* em relação à versão *Pipeline* se dá devido a necessidade de mais DSPs para processar o maior número de dados disponíveis em um dado momento.

Outro resultado interessante é o desempenho do *loop* mais interno do código (Algoritmo 1) e seu intervalo de iniciação, presentes na Tabela 2. A **Latência** é o número de ciclos de *clock* necessários para que uma iteração de um `loop` seja concluída, no caso, o *loop* responsável pelo cálculo de interseção entre um raio e um triângulo; e o **Intervalo de Iniciação** corresponde ao número de ciclos de *clock* necessários para que a próxima iteração possa ser iniciada. Os resultados mostram que ao adicionar a diretiva `ARRAY_PARTITION`, a latência total do *loop* aumentou, porém isto permitiu que o intervalo de iniciação fosse reduzido de 5 ciclos para 3. Os impactos desta mudança serão avaliados nas Subseções 5.2.1 e 5.2.2, que tratam da análise de desempenho dos aceleradores.

Figura 5. Utilização de Recursos da FPGA por cada versão do acelerador com o máximo possível de aceleradores instanciados.

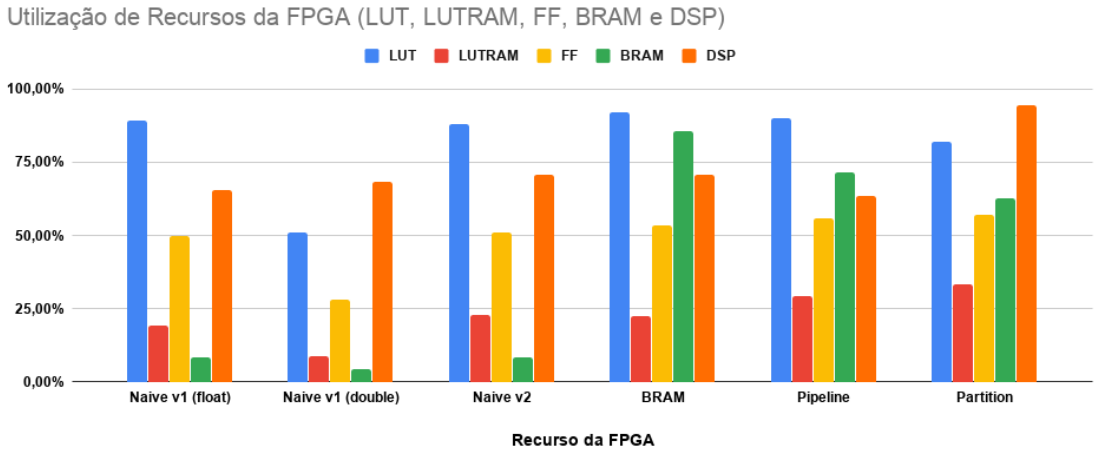


Tabela 2. Latência e intervalo de iniciação do *Pipeline*

Métrica (ciclos de *clock*)	***Pipeline***	***Array Partition***
Latência	120	202
Intervalo de Iniciação	5	3

5.2. Tempos de Execução

Nesta subseção serão descritos os resultados obtidos com relação ao tempo de execução de cada versão do acelerador, da CPU embarcada na placa, e dos *benchmarks* assim como as conclusões tiradas a partir dos mesmos. As execuções serão feitas renderizando imagens com resoluções 200×200, 400×400, 600×600, 800×800 e 1000×1000, em uma cena contendo 2.000 triângulos. Também foram comparados os tempos de execução da implementação do Algoritmo 1 em C++ para o processador ARM, sendo sua versão paralela implementada com a biblioteca OpenMP. O laço paralelizado na versão CPU é o laço mais externo, que itera pelos raios. Vale notar que a transferência de dados entre a memória RAM e as BRAMs da FPGA também faz parte dos tempos apresentados.

5.2.1. Tempos de execução para os aceleradores *single-core*

A Tabela 3 apresenta a média aritmética de 30 execuções de todos os aceleradores propostos para renderizar imagens com os modelos descritos anteriormente. É importante destacar que neste experimento apenas uma instância de cada acelerador foi executada, o que foi chamado de *single-core*.

Nestes tempos de execução é possível notar, primeiramente, que o processador ARM se manteve mais rápido do que todos os aceleradores anteriores à introdução da técnica de *pipelining*, o que se deve ao baixo uso da hierarquia de memória provida pelos recursos da FPGA. Como um processador de propósito geral, o ARM possui uma hierarquia de memória com caches L1 e L2 além de uma frequência da operação de 6,5x maior do que a da FPGA. Estas características em conjunto com a organização sequen-

Tabela 3. Tempos de execução (segundos) dos aceleradores utilizando apenas uma instância.

	ARM	Naive v1		Naive v2	BRAM	Pipeline	Partition
		float	double				
200x200	22,53	135,15	148,00	98,39	60,87	14,18	11,22
400x400	89,84	540,03	591,69	391,84	241,57	55,63	44,66
600x600	201,63	1215,22	1331,16	880,79	542,76	124,91	100,49
800x800	360,33	2160,85	2365,24	1566,99	962,36	221,33	178,62
1000x1000	565,43	3373,33	3694,86	2447,34	1513,89	358,50	279,15

cial dos dados na memória (que, pelo princípio da localidade espacial de *cache*, reduz o número de *cache-misses*), explicam o motivo dos aceleradores iniciais não serem capazes de providenciar um desempenho melhor do que o processador. Podemos ver, como exemplo, a queda no tempo de execução da versão *Naive v2* para a versão *BRAM* devido, simplesmente, a um melhor uso destes componentes.

Por outro lado, as versões *Pipeline* e *Array Partition*, obtiveram um desempenho não só comparável, como consideravelmente melhor do que o processador ARM, superando inclusive a diferença causada pela menor frequência. Isto acontece pelo fato de que múltiplas iterações do *loop* mais interno do algoritmo (*loop* dos triângulos) estão sendo executadas simultaneamente, aumentando assim a taxa de entrega do *loop* mais interno do acelerador. O impacto de uma maior taxa de entrega fica ainda mais evidente ao compararmos as versões *Pipeline* e *Array Partition*. A latência de iteração do *loop* de triângulos em número de ciclos é maior na versão *Array Partition* (Tabela 2 na Subseção 5.1), o que deveria, teoricamente, tornar o acelerador mais lento. Porém, isto não ocorre pois o maior particionamento nas *BRAM*s permite que os dados necessários para múltiplas iterações estejam disponíveis simultaneamente, o que é responsável pela queda no intervalo de iniciação do *loop*.

5.2.2. Tempos de execução para os aceleradores *multicore*

Esta subseção analisa os tempos de execução dos aceleradores *multicore* e da implementação ARM *double-core*. A quantidade de instâncias para cada acelerador está destacada na Tabela 1. Foram utilizados os modelos descritos na Subsecão 5.2.

Os tempos de execução obtidos pelas versões *multicore* estão descritos na Tabela 4. Quando se compara os tempos das versões *multicore* com os seus respectivos tempos das versões *single-core* fica claro a melhora de desempenho para todos os aceleradores. A versão *BRAM*, por exemplo, reduz mais de 5x seu tempo, ficando em geral um pouco melhor do que o da versão ARM *double-core*.

As versões *Pipeline* e *Partition*, por sua vez, foram as versões que obtiveram o pior desempenho quando se compara o tempo *single-core* com o *multicore* em relação a quantidade de instâncias executadas, reduzindo, aproximadamente, 3x e 2x. Uma possível causa para este fato é o congestionamento do barramento da memória DRAM e das BRAMs na FPGA, visto que o tempo de processamento dos blocos na memória se tornou consideravelmente mais rápido, causando a necessidade de mais acessos simultâneos a estes

recursos pelas instâncias sintetizadas. Por outro lado, podemos ver que conforme as imagens se tornam maiores, os *speed-ups* das versões *Pipeline* e *Partition* também crescem, sugerindo que conforme o custo computacional se torna mais intensivo, os *speed-ups* destas versões tendem a melhorar. Mesmo assim, as versões *Pipeline* e *Array Partition* obtiveram desempenho aproximadamente 2,5x melhor do que o processador ARM para o modelo 1000×1000, apesar da menor frequência de *clock* da FPGA.

Tabela 4. Tempos de execução (segundos) dos aceleradores utilizando o número máximo de instâncias para cada versão.

	ARM	**Naive v1**		**Naive v2**	**BRAM**	**Pipeline**	**Partition**
		float	**double**				
200x200	11,38	25,30	75,08	17,88	11,47	5,39	5,30
400x400	45,38	97,28	300,25	68,44	43,19	18,92	19,46
600x600	101,98	217,52	675,43	153,05	96,13	41,95	42,96
800x800	181,71	386,37	1199,93	271,69	172,62	74,08	76,07
1000x1000	284,81	613,95	1874,84	530,20	375,24	111,50	114,98

O *speed up* da implementação ARM é bastante relevante (até 1,99), mostrando que a divisão de tarefas é feita de forma eficiente, aproveitando quase que totalmente a natureza embaraçosamente paralela do algoritmo.

5.2.3. Comparação com os *Benchmarks*

Para uma melhor comparação do desempenho dos aceleradores, duas versões de *Benchmark* foram implementadas. A primeira versão, *Benchmark BRAM*, é a referência de desempenho para todas as versões implementadas sem o uso da técnica de *pipelining*, enquanto a versão *Benchmark Pipeline*, será utilizada como referência para as versões implementadas com o uso da técnica.

A comparação entre os aceleradores propostos e os *benchmarks* desenvolvidos neste trabalho foi realizada através da renderização de uma imagem de resolução 100×100 e 2.000 triângulos, que é o tamanho do modelo que cabe na memória BRAM da FPGA. Como já foi explicado na Subseção 4.6, a principal característica dos *benchmarks* é que todo o modelo 3D deve caber na BRAM para evitar a sobrecarga de comunicação ao longo da execução do algoritmo.

A Tabela 5 apresenta os tempos de execução (segundos) e coeficientes de variação da implementação ARM, dos aceleradores até a versão *BRAM* e da versão *Benchmark BRAM*. Os resultados mostram que até mesmo a versão *Benchmark BRAM* não foi capaz de obter um desempenho superior ao processador ARM, mostrando que apenas o uso mais eficiente dos componentes de memória BRAM não é suficiente para superar a diferença na frequência de operação e as vantagens da cache do processador ARM.

Por outro lado, o maior e melhor uso da memória *BRAM* proporcionou um ganho considerável de desempenho em relação às versões *Naive*, chegando a uma aceleração de até 2,8. Por fim, é importante destacar que o tempo de execução da versão *BRAM* foi muito próximo do tempo do *Benchmark*, que, como já explicado anteriormente, é

um limite inferior (inalcançável) de tempo que não considera o custo de transferência de dados entre o ARM e a FPGA. O que demonstra a eficiência da versão *BRAM* proposta.

Tabela 5. Comparação dos aceleradores com o *Benchmark BRAM*.

	ARM	Naive v1		Naive v2	BRAM	Benchmark
		Double	Float			
Tempo	5,67	37,04	32,52	24,50	15,29	13,09
Coef. de Variação	0,11%	0,01%	0,48%	0,59%	0,67%	0,93%

A Tabela 6, por sua vez, mostra os tempos de execução e os coeficientes de variação das versões *Pipeline* e *Array Partition* (*Partition* na tabela), e do *benchmark pipeline* referente às mesmas, além de repetir os valores da versão ARM para facilitar a comparação.

Em comparação com o desempenho da versão *BRAM*, os tempos de execução das versões *Pipeline* e *Array Partition* ficaram um pouco mais distante do tempo obtido pelo *benchmark Pipeline*. Essa diferença era esperada, uma vez que ao reduzirmos o tempo de processamento dos dados já armazenados, o tempo extra (sobrecarga) necessário para copiar um novo bloco de triângulos passou a ser uma fração mais significativa do total. Mesmo assim, essas versões produziram resultados bastante significativos, uma vez que o resultado obtidos pelo *benchmark* é um limite inferior inalcançável.

Tabela 6. Comparação dos aceleradores com o *Benchmark Pipeline*.

	ARM	Pipeline	Partition	Benchmark Pipeline
Tempo	5,67	3,69	3,28	1,68
Coef. de Variação	0,11%	0,25%	0,57%	0,98%

6. Conclusão

Por serem equipamentos de baixo custo energético e com alto potencial de poder computacional as arquiteturas reconfiguráveis, como as FPGAs, vem sendo cada vez mais utilizadas, inclusive para computação de alto desempenho.

Com o objetivo de aproveitar todo o potencial dessas arquiteturas este trabalho apresentou diferentes estratégias de otimização para o cálculo de interseção raio-triângulo, que é a parte mais custosa da técnica de *ray tracing*. Além disso, dois *benchmarks* foram desenvolvidos para avaliar o desempenho dos aceleradores propostos.

Os resultados obtidos mostraram que a versão sequencial mais rápida do acelerador foi ≈ 2 vezes mais rápida que a versão para o processador ARM, enquanto a melhor versão paralela obteve um fator de aceleração de até 4,73x. Mais importante, os resultados obtidos ficaram muito próximos dos *benchmarks* propostos que tem a característica de serem um limite inferior de tempo sem a sobrecarga de transferência de dados para a memória BRAM, o que demonstra a eficiência dos aceleradores propostos.

Agradecimentos

Esta pesquisa é financiada pelos projetos CNPq 426729/2018-8 e FAPDF 00193-00002139/2018-79. Por último, mas não menos importante, os autores gostariam de

agradecer ao Programa Universitário da Xilinx pela doação das licenças que permitiram o desenvolvimento deste trabalho.

Referências

Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006. ISBN 0471720925.

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

Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Computing Surveys (CSUR)*, 50(4):58, 2017.

Digilent. *PYNQ-Z1 Board Reference Manual*. Digilent Inc., 2017.

J. Fender and J. Rose. A high-speed ray tracing engine built on a field-programmable system. In *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*, pages 188–195, Dec 2003. doi: 10.1109/FPT.2003.1275747.

Andrew S Glassner. An introduction to ray tracing. chapter 1, pages 1–30. Elsevier, 1989.

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

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

Tomas Möller and Ben Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

A. S. Nery, N. Nedjah, and F. M. G. França. A parallel architecture for ray-tracing. In *2010 IEEE Latin American Symp. on Circuits and Systems*, pages 77–80, Feb 2010. doi: 10.1109/LASCAS.2010.7410224.

W. C. Park, Jae ho Nah, J. S. Park, Kyung-Ho Lee, Dong-Seok Kim, Sang-Duk Kim, J. H. Park, Cheong-Ghil Kim, Yoon-Sig Kang, Sung-Bong Yang, and Tack-Don Han. An fpga implementation of whitted-style ray tracing accelerator. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 187–187, Aug 2008. doi: 10.1109/RT.2008.4634650.

Adrianno Sampaio, Alexandre Sena, and Alexandre Nery. Um sistema heterogêneo embarcado para aceleração de interseção raio-triângulo. In *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 394–405, Porto Alegre, RS, Brasil, 2019. SBC. doi: 10.5753/wscad.2019.8685. URL `https://sol.sbc.org.br/index.php/wscad/article/view/8685`.

Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. 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, 2004. ACM. ISBN 3-905673-15-0. doi: 10.1145/1058129.1058143. URL `http://doi.acm.org/10.1145/1058129.1058143`.

Tim Todman and Wayne Luk. Reconfigurable designs for ray tracing. In *null*, pages 300–301. IEEE, 2001.

P. Wilson. *Design Recipes for FPGAs: Using Verilog and VHDL*. Elsevier Science, 2011.

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

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