Uma Vetorização Eficiente para o Algoritmo de Interseção Raio-Triângulo nas Arquiteturas Multicore

Alexandre C. Sena[1], **Adrianno A. Sampaio**[1], **Aline P. Nascimento**[2], **Alexandre S. Nery**[3]

[1]Instituto de Matemática e estatística – Universidade do Estado do Rio de Janeiro
Rio de Janeiro – RJ – Brasil

[2]Instituto de Computação – Universidade Federal Fluminense
Niterói – RJ – Brasil

[3]Departamento de Engenharia Elétrica – Universidade de Brasília (UnB)
Brasília – DF – Brasil

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

Resumo. *O cálculo de interseção raio-triângulo é a parte mais custosa do algoritmo de* Ray Tracing*, que é uma das técnicas mais utilizadas para síntese de imagens 3D por renderizar cenas muito próximas da realidade. Assim, o objetivo deste trabalho é propor e implementar uma vetorização eficiente para, em conjunto com o paralelismo, explorar o potencial das arquiteturas multicore. Ao invés de uma abordagem simplificada não escalável, a estratégia proposta calcula a interseção de um raio com vários triângulos da cena ao mesmo tempo. Resultados experimentais mostraram que a versão vetorizada paralela aproveitou as instruções vetorizadas e os múltiplos núcleos disponíveis conseguindo um desempenho até* 257 *vezes melhor do que a versão sequencial otimizada.*

1. Introdução

Ray Tracing é um algoritmo de Computação Gráfica usado para síntese de imagens tridimensionais [Glassner 1989]. Por conseguir renderizar cenas muito próximas da realidade essa técnica é muito utilizada no desenvolvimento de filmes.

Um *Ray Tracer* funciona a partir da simulação da interação entre raios de luz e os objetos de uma cena tridimensional. Em sua forma mais simples, o algoritmo de *Ray Tracing* é altamente custoso [Whitted 1980]. Dada uma câmera virtual de $A \times L$ pixels, sendo A a altura e L a largura da imagem que se deseja capturar, é lançado um raio (primário) para cada pixel e são realizados cálculos de interseção para todos os N objetos existentes na cena, o que requer a execução de pelo menos $A \times L \times N$ operações de interseção. Além disso, para cada interseção encontrada podem ser gerados novos raios (secundários) que também precisarão ser testados contra os N objetos existentes na cena, o que torna esse algoritmo muito custoso computacionalmente.

Logo, para extrair melhor desempenho do algoritmo é preciso fazer bom uso dos recursos disponíveis nas arquiteturas *multicore* (e.g. `Intel Xeon`). Por exemplo, usar eficientemente as instruções `SIMD` (*Single-Instruction Multiple-Data*) em conjunto com os múltiplos núcleos disponíveis [Diaz et al. 2012]. Uma grande vantagem dessas arquiteturas, além do seu alto poder computacional, é estarem disponíveis na maioria dos ambientes de desenvolvimento. Nesse contexto, o objetivo deste trabalho é propor uma

vetorização eficiente para o cálculo de interseção raio-triângulo, que é a parte mais custosa de um *Ray Tracing*. Mais especificamente, uma vetorização para as novas instruções `AVX-512` é implementada. Os resultados obtidos mostram que a vetorização proposta é eficiente, sendo na média $\approx 8,4$ vezes melhor que a versão sequencial otimizada. Mais importante, seu uso em conjunto com os múltiplos núcleos disponíveis mostram que é possível explorar todo o potencial dessas arquiteturas, atingindo até 257x de *speedup*.

O restante deste trabalho está organizado da seguinte forma: na Seção 2 são apresentados os trabalhos relacionados; a Seção 3 apresenta a algoritmo de interseção raio-triângulo; em seguida, na Seção 4, a vetorização proposta é descrita; a análise experimental é apresentada na Seção 5; por fim, as conclusões são apresentados na Seção 6.

2. Trabalhos Relacionados

Sabe-se que um dos maiores desafios do algoritmo de *Ray Tracing* está no cálculo rápido de operações de interseção para atingir desempenho em tempo real, ou seja, igual ou superior a 30 quadros por segundo. Neste sentido, o algoritmo de *Ray Tracing* tem sido objeto de estudo ao longo das últimas décadas, em especial no que diz respeito à implementação de arquiteturas dedicadas em *hardware* para aceleração de sua execução [Woop et al. 2006, Nery et al. 2010].

Ainda no tocante à construção de arquiteturas dedicadas em *hardware*, o trabalho de [Okmen 2011] expande o conjunto de instruções de um microprocessador `VLIW` com instruções `SIMD` de ponto flutuante. Porém, tal microprocessador é voltado para aplicações de baixo consumo de energia e, logo, a extensão ainda não foi capaz de entregar desempenho satisfatório quando comparada a implementações do mesmo algoritmo em `CPU` e `GPU`.

Quanto ao uso de instruções `SIMD` disponíveis em arquiteturas paralelas, o trabalho apresentado em [Wald et al. 2008] investiga diferentes formas de empacotamento das operações do *Ray Tracing* em instruções `SIMD`. Outro trabalho apresentado em [Havel and Herout 2010] propõe um algoritmo novo para o cálculo da interseção raio-triângulo. Uma das abordagens utilizadas para acelerar o algoritmo é a vetorização de 4 produtos escalares através das instruções `SSE4` disponíveis na época. O problema desta abordagem é que ela não é escalável, ou seja, ela não aproveita a maior quantidade de registradores `SIMD` disponíveis nos novos conjuntos de instruções como, por exemplo, `AVX-2` e `AVX-512`.

Por fim, o trabalho apresentado em [Sena et al. 2017] apresenta de forma didática os passos necessários para vetorizar uma parte do algoritmo de leilão usando `Intel intrinsics`.

3. Algoritmo de *Ray Tracing*

O algoritmo de *Ray Tracing* tem o objetivo de simular a interação entre raios de luz e os objetos de uma cena tridimensional. Uma descrição desse algoritmo pode ser visualizada na Figura 1. Basicamente, o algoritmo percorre todos os raios (i.e. um raio para cada pixel da imagem), como pode ser visto na Linha 4, e verifica como cada raio influencia nos objetos da cena 3D (Linhas 15 a 33) aplicando o Algoritmo de Möller-Trumbore [Möller and Trumbore 2005].

```
computeIntersections(vector<float> rayData, vector<int> triIds, vector<float> triData) {
 vector<int> outIds(numRays);
 vector<float> outInter(numRays);
 for(int ray=0; ray < rayData[0].size(); ray++){
  outIds[ray] = -1; outInter[ray] = 1.0e+9;
  VEC3(origin); VEC3(direction);
  origem[0] = rayData[0][ray]; origem[1] = rayData[1][ray]; origem[2] = rayData[2][ray];
  direcao[0] = rayData[3][ray]; direcao[1] = rayData[4][ray]; direcao[2] = rayData[5][
      ↪ ray];
  for(tri=0; tri < triangleData[0].size(); tri++){
    VEC3(v0); VEC3(v1); VEC3(v2);
    v0[0] = triData[0][tri]; v0[1] = triData[1][tri]; v0[2] = triData[2][tri];
    v1[0] = triData[3][tri]; v1[1] = triData[4][tri]; v1[2] = triData[5][tri];
    v2[0] = triData[6][tri]; v2[1] = triData[7][tri]; v2[2] = triData[8][tri];

    VEC3(edge1); VEC3(edge2); VEC3(h);
    SUB(edge1, v1, v0); SUB(edge2, v2, v0);
    CROSS(h, direcao, edge2);
    float a = DOT(edge1, h);
    if (fabs(a) < EPSILON) continue;

    float f = 1.0 / a;
    VEC3(s); SUB(s, origem, v0);
    float u = f * DOT(s, h);
    if (u < 0.0 || u > 1.0) continue;

    VEC3(q); CROSS(q, s, edge1);
    float v = f * DOT(direcao, q);
    if (v < 0.0 || u + v > 1.0) continue;

    float t = f * DOT(edge2, q);
    if (t < outInter[ray] && t > EPSILON){
      outIds[ray] = triIds[tri], outInter[ray] = t;
    }
  }
 }
 return std::make_pair(outIds, outInter);
}
```

Figura 1. Código da função de cálculos de interseção raio-triângulo

3.1. Algoritmo de Möller-Trumbore

O algoritmo de Möller-Trumbore tem como objetivo realizar o cálculo de um ponto de interseção entre um raio R e um triângulo em um espaço tridimensional. Podemos definir um triângulo como um conjunto de três pontos tridimensionais tal como $T = (V_0, V_1, V_2)$, onde V_0, V_1, V_2 são vetores tridimensionais indicando os vértices de T. Um dado ponto na superfície de um triângulo pode ser denotada por $T(u, v) = (1-u-v)V_0 + uV_1 + vV_2$, onde u e v são as coordenadas baricêntricas do ponto em relação aos vértices de um triângulo. Para que um ponto esteja contido na superfície de um triângulo, suas coordenadas baricêntricas devem satisfazer as seguintes condições: $u \geq 0, v \geq 0, u + v \leq 1$.

Da mesma forma, um raio matemático pode ser definido a partir de um ponto de origem $O = (O_x, O_y, O_z)$ e um vetor direção $D = (D_x, D_y, D_z)$ de forma que um ponto neste raio corresponde à $R(t) = O + tD$, onde t é a distância entre o ponto e a origem do raio. Nesta equação, t assume um valor positivo para os pontos "à frente" da origem do raio, ou seja, estão posicionados no sentido para onde a direção D do raio aponta, e valores negativos para os pontos "atrás" da origem do raio.

O algoritmo tem como conceito básico aplicar uma transformação linear ao raio e ao triângulo em questão de forma que os pontos deste triângulo sejam $V_0 = (0, 0, 0), V_1 =$

$(1, 0, 0)$ e $V_2 = (0, 1, 0)$. Em seguida o algoritmo obtém o ponto de interseção entre o raio transformado e o plano XY. Sabendo este ponto de interseção $P_{intersect} = (x, y, 0)$ é possível encontrar as coordenadas baricêntricas em relação ao triângulo T. Com isto, o algoritmo verifica se as coordenadas baricêntricas calculadas correspondem a um ponto no triângulo transformado.

Este processo nos leva à equação 1, que calcula a distância t e as coordenadas baricêntricas u e v. Onde denotamos $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$, $T = O - V_0$.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2).E_1} \begin{bmatrix} (T \times E_1).E_2 \\ (D \times E_2).T \\ (T \times E_1).D \end{bmatrix} = \frac{1}{P.E_1} \begin{bmatrix} Q.E_2 \\ P.T \\ Q.D \end{bmatrix} \tag{1}$$

Além disso, também estão denotados os vetores intermediários $P = (D \times E_1)$ e $Q = (T \times E_1)$ no lado direito da equação.

Na Figura 2 estão definidas as macros auxiliares para realizar operações com vetores tridimensionais. Foram definidas as operações `VEC3` (Linha 5), `ASSIGN` (Linha 6), `DOT` (Linha 7), `CROSS` (Linha 8) e `SUB` (Linha 12) que servem respectivamente para, alocação, atribuição, produto escalar, produto vetorial e subtração.

```
01 | #define EPSILON 1.0e-6
02 | #define INFINITY 1.0e9
03 | #define TRIANGLE_ATTR_NUMBER 9
04 | #define RAY_ATTR_NUMBER 6
05 | #define VEC3(NAME) float NAME[3]
06 | #define ASSIGN(VL, VR) (VL)[0] = (VR)[0]; (VL)[1] = (VR)[1]; (VL)[2] = (VR)[2]
07 | #define DOT(V1, V2) (V1[0]*V2[0] + V1[1]*V2[1] + V1[2]*V2[2])
08 | #define CROSS(VR, V1, V2) \
09 |     VR[0] = V1[1] * V2[2] - V1[2] * V2[1], \
10 |     VR[1] = V1[2] * V2[0] - V1[0] * V2[2], \
11 |     VR[2] = V1[0] * V2[1] - V1[1] * V2[0]
12 | #define SUB(VR, V1, V2)  \
13 |     VR[0] = V1[0] - V2[0];  \
14 |     VR[1] = V1[1] - V2[1];  \
15 |     VR[2] = V1[2] - V2[2]
```

Figura 2. Macros C++ auxiliares para operações com vetores

O cálculo dos termos t, u e v definidos na Equação 1 podem ser vistos da linha 23 a 33. As verificações de consistência, como as das coordenadas baricêntricas, podem ser visualizadas nas linhas 19, 24 e 28 do algoritmo. Estes comandos condicionais servem para evitar cálculos desnecessários quando é possível garantir que não houve uma interseção. Na Linha 30 é verificado se a distância de interseção t é positiva e menor do que as interseções anteriores. Se este for o caso, o ponto de interseção mais próximo atingido pelo raio `ray` é atualizado (linha 32).

4. Vetorização nas Arquiteturas `Xeon`

Para utilizar o potencial das arquiteturas `Xeon` e `Xeon Phi`, além de utilizar os vários núcleos disponíveis, é necessário utilizar as instruções vetorizadas onde a mesma operação pode ser realizada em um conjunto de dados. Enquanto que a primeira versão dessa tecnologia, chamada de `MMX`, disponibilizava registradores de apenas 64 *bits*, a versão atual, chamada de `AVX-512`, disponibiliza registradores de 512 *bits*.

A vetorização é a conversão de uma implementação escalar para um processo vetorial [Mark-Sabahi 2012]. Esse processo é essencial para extrair desempenho nessas arquiteturas, especialmente nos novos processadores `Xeon` que são capazes de realizar até 16 operações de ponto flutuante simultaneamente. Em alguns casos, para códigos simples ou bem otimizados, o compilador é capaz de vetorizar o programa. Porém, para que isso aconteça o compilador tem que ser capaz de identificar a ausência de dependência de dados nas operações e que o acesso a memória seja contíguo [Mark-Sabahi 2012].

Uma vez que o compilador não é capaz de vetorizar o código duas alternativas podem ser adotadas: vetorização explícita (diretivas `OpenMP`) ou programação com instruções `SIMD` (`intrinsics`) [Wende et al. 2016]. A vetorização explícita consiste em fornecer informações adicionais (através de diretivas) para ajudar o compilador a vetorizar o código. Por sua vez, `intrinsics` é uma programação de baixo nível que utiliza as instruções `SIMD`. Embora a vetorização explícita seja menos complexa, o seu desempenho depende não somente do uso preciso das diretivas, mas também da maneira como elas foram implementadas. Por outro lado, apesar da programação `intrinsics` ser mais complexa, ela permite explorar todo o potencial das instruções `SIMD`.

A subseção seguinte apresenta a versão vetorizada para o algoritmo de *Ray Tracing* proposta neste trabalho. Com o objetivo de aproveitar todo potencial das arquiteturas `Xeon`, foi implementada uma versão vetorizada utilizando a programação `intrinsics`.

4.1. Vetorização do Algoritmo de *Ray Tracing*

Conforme apresentado na Subseção 3.1, o cálculo de interseção raio-triângulo consiste basicamente em operações com vetores tridimensionais tais como produto escalar e produto vetorial. Ao se observar as macros definidas na Figura 2, é possível verificar que elas são compostas de operações que são independentes uma das outras. Por exemplo, a operação chamada de SUB é composta de três subtrações independentes. Assim, uma maneira simplificada de vetorizar o algoritmo de *Ray Tracing* seria, para cada par (raio-triângulo) realizar essas operações simultaneamente. Porém, essa estratégia limita o desempenho da vetorização, além de não ser escalável.

Como já foi mencionado anteriormente, as novas arquiteturas disponibilizam registradores de 512 bits, o que permite a execução de até 16 instruções de ponto flutuante por vez. Porém, usando a estratégia de vetorização simplificada anteriormente explicada, a operação SUB seria executada usando apenas 3 instruções de ponto flutuante por vez. Além disso, caso novas arquiteturas disponibilizem registradores `SIMD` ainda maiores, o desempenho dessa estratégia continuará limitado a executar apenas 3 instruções por vez, o que torna essa estratégia não escalável.

Assim, a vetorização proposta neste trabalho adota outra abordagem. Conforme pode ser visto na Figura 1, o cálculo de interseção de cada raio com cada triângulo da cena é independente. Ou seja, eles podem ser calculados simultaneamente. Logo, a estratégia de vetorização proposta neste trabalho aproveita essa característica da aplicação, calculando a interseção de um único raio com vários triângulos da cena ao mesmo tempo.

Enquanto que os raios do algoritmo são definidos através de sua origem e direção compostas de coordenadas tridimensionais, a cena é composta de triângulos definidos através de seus 3 vértices também compostos de coordenadas tridimensionais (como explicado na Seção 3). Assim, para conseguir carregar os dados de vários triângulos nas

variáveis vetorizadas optou-se por usar uma variável para cada coordenada de cada componente. Por exemplo, o vértice $v0$ do triângulo que é composto das coordenadas x, y e z foi dividido em três variáveis vetorizadas chamadas de $v0x$, $v0y$, $v0z$.

A base do cálculo de interseção raio-triângulo são as operações matemáticas nos vetores tridimensionais. A Figura 3 apresenta a vetorização das operações de subtração, produto vetorial e produto escalar.

```
01 | subVet(rx, ry, rz, op1x, op1y, op1z, op2x, op2y, op2z) {
02 |   rx = SUBV(op1x,op2x);
03 |   ry = SUBV(op1y,op2y);
04 |   rz = SUBV(op1z,op2z);
05 | }
06 | crossVet(rx, ry, rz, op1x, op1y, op1z, op2x, op2y, op2z) {
07 |   rx = MULT(op1y,op2z);
08 |   aux = MULT(op1z,op2y);
09 |   rx = SUBV(rx,aux);
10 |   ry = MULT(op1z,op2x);
11 |   aux = MULT(op1x,op2z);
12 |   ry = SUBV(ry,aux);
13 |   rz = MULT(op1x,op2y);
14 |   aux = MULT(op1y,op2x);
15 |   rz = SUBV(rz,aux);
16 | }
17 | dotVet(r, op1x, op1y, op1z, op2x, op2y, op2z) {
18 |   r = MULT(op1x,op2x);
19 |   aux = MULT(op1y,op2y);
20 |   r = SOMA(r,aux);
21 |   aux = MULT(op1z,op2z);
22 |   r = SOMA(r,aux);
23 | }
```

Figura 3. Subtração, Produto Vetorial e Produto Escalar Vetorizados

A operação de subtração, função `subVet`, recebe como parâmetros variáveis vetorizadas onde são realizadas 3 subtrações (linhas 2 a 4). É importante ressaltar que cada subtração (representada pelo mnemônico `SUBV`) realiza a operação em uma variável vetorizada, o que significa que são realizadas N subtrações de uma vez, onde N depende do tipo da arquitetura e do tipo de dado. Por exemplo, para instruções de ponto flutuante de precisão simples ($float = 32bits$) em uma arquitetura compatível com instruções `AVX-512` ($512bits$) é possível executar $\frac{512}{32} = 16$ instruções simultaneamente.

As funções crossVet (linhas 7 a 15) e dotVet (linhas 18 a 22), por sua vez, vetorizam as operações de produto vetorial e produto escalar seguindo o mesmo princípio da subtração. O código principal da vetorização do cálculo de interseção raio-triângulo pode ser visto na Figura 4. O código inicia no comando for da linha 3 que percorre todos os raios da imagem. Em seguida, da linha 4 a 9 são inicializadas as variáveis vetorizadas que definem cada raio (origem e direção). Porém, conforme já foi explicado anteriormente, para facilitar o acesso a cada coordenada dessas variáveis, o código utiliza uma variável vetorizada para cada coordenada (e.g. origem = origemx, origemy e origemz). As variáveis origem e direção são inicializadas com os dados do raio corrente. Ou seja, as 16 posições são inicializadas com os dados de apenas um único raio, pois a abordagem da vetorização proposta é calcular a interseção de um mesmo raio com vários triângulos.

O comando for da linha 10 percorre os triângulos que formam a cena 3D. Porém, ao invés de iterar uma unidade por vez, o comando for percorre os triângulos de 16

```
computeIntersectionsSIMD( vector<float> rayData, vector<int> triIds, vector<float>
    ↪ triData){
vector<int> outIds(numRays);vector<float> outInter(numRays);
for(int ray=0; ray < rayData[0].size(); ray++){
 outIds[ray] = -1; outInter[ray] = 1.0e+9;
 tFinal = SET(1.0e+9); triFinal = ISET(-1);
 origemx = SET(rayData[0][ray]);origemy = SET(rayData[1][ray]);
 origemz = SET(rayData[2][ray]);
 direcaox = SET(rayData[3][ray]);direcaoy = SET(rayData[4][ray]);
 direcaoz = SET(rayData[5][ray]);
 for(i=0; i < triangleData[0].size(); i += 16){
  v0x = LOAD((&(triData[0][0])+i));v0y = LOAD((&(triData[1][0])+i));
  v0z = LOAD((&(triData[2][0])+i));
  v1x = LOAD((&(triData[3][0])+i));v1y = LOAD((&(triData[4][0])+i));
  v1z = LOAD((&(triData[5][0])+i));
  v2x = LOAD((&(triData[6][0])+i));v2y = LOAD((&(triData[7][0])+i));
  v2z = LOAD((&(triData[8][0])+i));
  subVet(edge1x, edge1y, edge1z, v1x, v1y, v1z, v0x, v0y, v0z);
  subVet(edge2x, edge2y, edge2z, v2x, v2y, v2z, v0x, v0y, v0z);
  crossVet(hx,hy,hz,direcaox,direcaoy,direcaoz,edge2x,edge2y,edge2z)
  dotVet(a,edge1x, edge1y, edge1z, hx, hy, hz); aAbs = ABSV(a);
  mask = CMP(aAbs, 1.0e-6, _CMP_GE_OS);
  if(mask){
   subVet(sx, sy, sz, origemx, origemy, origemz, v0x, v0y, v0z);
   dotVet(u,sx, sy, sz, hx, hy, hz);
   u = MULT(f,u);
   mask2 = CMP(u, 0.0,_CMP_GE_OS); mask = mask & mask2;
   mask2 = CMP(u, 1.0,_CMP_LE_OS); mask = mask & mask2;
   if(mask){
    crossVet(qx,qy,qz,sx,sy,sz,edge1x,edge1y,edge1z);
    dotVet(v,directionx, directiony, directionz, qx, qy, qz);
    v = MULT(f,v);
    mask2 = CMP(v, 0.0,_CMP_GE_OS); mask = mask & mask2;
    aux = SOMA(u,v);
    mask2 = CMP(aux, 1.0,_CMP_LE_OS); mask = mask & mask2;
    if(mask){
     dotVet(t,edge2x,edge2y,edge2z, qx, qy, qz); t = MULT(f,t);
     mask2 = CMP(t, 1.0e-6,_CMP_GT_OS); mask = mask & mask2;
     mask2 = CMP(t, tFinal,_CMP_LT_OS); mask = mask & mask2;
     if(mask){
      tFinal = MBLEND(mask,tFinal,t);
      vTri = ISET(i);
      triFinal = IMBLEND(mask,triFinal,vTri);}}}}
 }// fim do loop triangulos
 float auxt[16]; int auxTri[16];
 STORE (auxt,tFinal);ISTORE (auxTri,triFinal);
 float menort = 1.0e9;
 for (int i=0;i<chunk;i++){
  if (auxt[i] < menort){
   menort = auxt[i];
   outIds[ray] = auxTri[i] + i;
   outInter[ray] = auxt[i];}
 }
}// fim do loop raios
return std::make_pair(outIds, outInter);
}
```

Figura 4. Código Vetorizado da função de cálculos de interseção raio-triângulo

em 16, uma vez que as variáveis vetorizadas são capazes de armazenar 16 valores de ponto flutuante de precisão simples pois os registradores da arquitetura `Xeon` mais recente possuem tecnologia `AVX-512`. Assim, da linha 11 a 16 são inicializadas as variáveis vetorizadas com os 3 vértices de 16 triângulos por vez. Assim como foi feito com os dados dos raios, foi criado uma variável para cada coordenada de um vértice.

Em seguida, da mesma maneira que no algoritmo original (Figura 1), são chamados os procedimentos que executam as operações com vetores tridimensionais (subtração, produto escalar e produto vetorial) explicadas anteriormente. Na linha 21, o comando `CMP` compara se o valores absolutos armazenados na variável vetorizada a são significativos. Ou seja, se são maiores que um valor constante muito pequeno. O objetivo dessa comparação é evitar cálculos desnecessários. O resultado dessas 16 comparações são armazenados em $mask$, colocando o valor 1 se a comparação for verdadeira e 0 se for falsa. Caso as 16 comparações sejam falsas então o valor da variável $mask$ vai ser 0. Logo, o comando if da linha 22 será falso e a iteração corrente é terminada. Porém, basta que apenas uma das 16 comparações seja verdadeira para a execução prosseguir dentro do if.

O programa prossegue realizando outras operações vetoriais (linhas 23 a 25). As linhas 26 e 27 verificam se as condições das coordenadas baricêntricas são satisfeitas. Para isso, a máscara de $bits\ mask2$ recebe através do comando `CMP` 1 se o valor de $u \geq 0.0$ e 0 caso contrário. Em seguida, é feito uma operação `AND` entre as máscaras de $bits\ mask$ e $mask2$. Essa operação `AND` é muito importante pois, como estão sendo realizadas 16 cálculos ao mesmo tempo, ela atualiza o valor da variável $mask$ com as novas condições de maneira que se novos valores falsos (0) surgirem é possível que a variável $mask$ tenha se tornado 0 e a iteração possa terminar (ou seja, evita realizar operações vetoriais desnecessárias para os 16 cálculos de iteração raio-triângulo que estão sendo processados). O comando if da linha 28 segue o mesmo padrão do if anterior. Os passos da linha 29 a 39 seguem o mesmo padrão do que já foi explicado. As variáveis vetorizadas $tFinal$ e $triFinal$ foram inicializadas com os valores $1.0e+9$ e -1, respectivamente, na linha 7 do algoritmo. Nas linhas 40 e 42 elas são atualizadas com o valor calculado de t e o identificador do triângulo, que são respectivamente o valor da distância e a qual triângulo ela se refere. Para atualizar o valor de $tFinal$ é utilizado o comando `MBLEND`.

O comando `MBLEND`, linha 40, permite uma junção entre as variáveis $tFinal$ e t, apenas das posições que tiverem o valor 1 em $mask$. Ou seja, só serão atualizadas as posições da variável vetorizada $tFinal$ com o valor de t onde o *bit* correspondente na variável $mask$ tiver valor 1. Por sua vez, para atualizar a variável $triFinal$ primeiramente a variável vetorizada $vTri$ é inicializada com o valor corrente da variável i (for que percorre os triângulos), através do comando `ISET`. Em seguida, através do comando `IMBLEND`, apenas os campos da variável $triFinal$ onde a variável $mask$ for igual 1 são atualizados com o valor de $vTri$. A única diferença entre os comandos `MBLEND` e `IMBLEND` é que o primeiro trabalha com valores $float$ e o segundo com valores inteiros.

Ao terminar uma iteração do for mais interno (triângulos) as variáveis vetorizadas $tFinal$ contém 16 distâncias referentes ao cálculo de 16 interseções raio-triângulo realizadas ao mesmo tempo. A variável $triFinal$ contém o valor de i corrente. Porém, o resultado final deve conter apenas o triângulo mais próximo do raio (ao invés de 16 contidos na variável vetorizada). Logo, é necessário uma fase de redução que se inicia na linha 44. Os 16 valores das variáveis vetorizadas $tFinal$ e $triFinal$ são copiados para os

vetores $auxt$ e $auxTri$, através dos comandos STORE e ISTORE (linha 45). A varíavel menort que irá armazenar o menor valor é inicializada com uma valor muito alto (linha 46). O processo para achar a menor distância é muito simples, bastando percorrer as 16 posições do vetor e guardar sempre a de menor valor (linhas 47 a 52). Para encontrar o triângulo relativo a distância mais próxima é necessário acrescentar a posição armazenada durante a vetorização o valor do índice relativo a menor distância (linha 50).

5. Análise Experimental

Todos os experimentos foram realizados em uma máquina multicore NUMA composta de dois processadores Intel Xeon® Platinum 8160 com um total de 48 núcleos (frequência base de 2.10GHz) e 187GB de memória. Esta arquitetura disponibiliza instruções SIMD do tipo AVX-512 o que permite usar todo o potencial da vetorização proposta neste trabalho. Todas as versões do algoritmo de interseção raio-triângulo avaliada neste trabalho foram executadas 30 vezes e a média aritmética e o coeficiente de variação foram calculados. Os experimentos foram divididos em dois tipos: vetorização e vetorização+SIMD. As subseções a seguir apresentam os resultados obtidos.

5.1. Análise de Desempenho da Vetorização

Três versões do algoritmo de interseção raio-triângulo foram executadas: (i) Original; (ii) Otimizada; (iii) Vetorizada. A versão **Original** é similar ao algoritmo apresentado na Figura 1. A única diferença é que, ao invés de utilizar um vetor para cada uma das coordenadas dos vértice do triângulo e da origem e o destino dos raios, é utilizado um vetor para armazenar todos os vértices do triângulo, outro para os dados dos raios. O objetivo desta estrutura de dados é melhorar a localização dos dados (i.e. aumentar o acerto da cache). A versão **Otimizada** é exatamente igual ao algoritmo da Figura 1. Por fim, a versão proposta neste trabalho, chamada de **Vetorizada**, descrita na Subseção 4.1.

Todas as versões foram compiladas utilizando o compilador icc da Intel (versão 18.0.2) com otimizacão -qopt-report-file -O3 -xCORE-AVX512. O parâmetro -xCORE-AVX512 é usado para instruir o compilador a tirar proveito das instruções AVX-512 SIMD disponíveis nesta arquitetura. Por sua vez, o parâmetro -qopt-report-file gera em um arquivo um relatório com todas as otimizações que o compilador icc conseguiu efetuar. Enquanto que a versão **Original** não conseguiu nenhum benefício da vetorização, a versão **Otimizada** foi parcialmente vetorizada, de acordo com o relatório da compilação. Por sua vez, a versão **Vetorizada** foi totalmente vetorizada. Para avaliar as versões foram executados 17 modelos 3D com quantidades de raios e triângulos distintas. Os resultados obtidos podem ser vistos na Tabela 1. Para cada um dos modelos 3D são apresentados a quantidade de raios e triângulos, os tempos de execução das 3 versões avaliadas e o *speedup* da versão vetorizada em relação a versão original e a versão otimizada.

O tempo para achar a interseção raio-triângulo é proporcional a quantidade de raios (i.e. tamanho da imagem) e a quantidade de triângulos (i.e. quantidade de objetos da cena 3D). Observando os tempos obtidos para cada uma das versões, fica claro que o desempenho da versão vetorizada foi bem superior para todos os modelos executados. A razão para tal superioridade é o melhor aproveitamento das instruções SIMD.

O ganho de desempenho da versão proposta pode ser melhor avaliado através dos *speedups* da Tabela 1. Considerando a média aritmética dos

Tabela 1. Comparação das versões sequenciais original, otimizada e vetorizada

Modelos	Características		Tempos (segundos)			Speedup	
	Raios	Triângulos	Original	Otimizada	Vetorizada	Original	Otimizada
mod1	40.000	2.000	1,016	0,589	0,061	16,65	9,65
mod2	40.000	50.000	26,430	16,523	2,502	10,56	6,60
mod3	1.638.400	2.000	37,779	20,037	1,953	19,34	10,26
mod4	10.000	2.000	0,274	0,167	0,027	10,15	6,18
mod5	262.144	2.000	6,336	3,473	0,375	16,89	9,26
mod6	1.048.576	2.000	24,982	13,137	1,517	16,47	8,66
mod7	4.194.304	2.000	99,648	49,072	5,681	17,54	8,64
mod8	40.000	5.000	1,496	1,262	0,138	10,84	9,14
mod9	40.000	32.258	11,947	6,926	1,044	11,44	6,63
mod10	40.000	15.774	5,861	3,725	0,391	14,99	9,53
mod11	40.000	5.000	1,111	0,824	0,108	10,29	7,63
mod12	40.000	5.000	1,503	1,281	0,146	10,29	8,77
mod13	262.144	5.000	10,489	8,377	0,942	11,13	8,89
mod14	262.144	32.258	82,729	47,715	6,621	12,49	7,21
mod15	262.144	15.744	42,354	23,581	2,541	16,67	9,28
mod16	262.144	5.000	7,828	5,399	0,702	11,15	7,69
mod17	262.144	5.000	10,277	7,794	0,963	10,67	8,09

$speedup_{original}(\frac{TempoExec_{original}}{TempoExec_{vetorizado}})$ para todos os modelos, a versão vetorizada foi $\approx 13,4$ melhor do que a versão original. Por sua vez, ao avaliar a média aritmética do $speedup_{otimizado} = \frac{Tempo_{otimizado}}{Tempo_{vetorizado}}$, é possível observar claramente a superioridade da versão vetorizada ($\approx 8,4$ vezes melhor). Os resultados obtidos são bastante confiáveis com coeficiente de variação $< 2,5\%$.

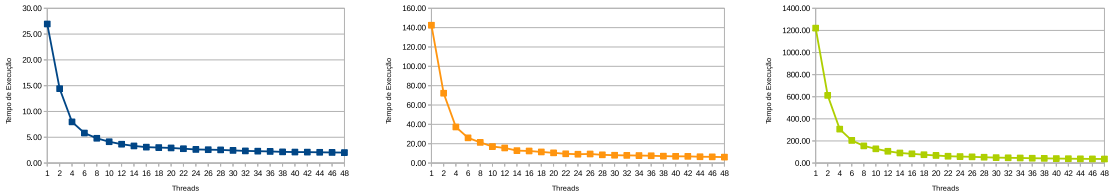
Analisando as razões para o alto desempenho da vetorização proposta, duas características podem ser destacadas: cálculo da interseção raio-triângulo de 16 pares simultaneamente e uso das instruções condicionais (linhas 22, 28, 35 e 39 da Figura 4).

5.2. Análise de Desempenho da Vetorização + Paralelismo

Para explorar o potencial das máquinas *multicore*, além de uma vetorização eficiente, é necessário utilizar os múltiplos núcleos disponíveis. Para isso, foi utilizado o modelo de programação paralela para memória compartilhada `OpenMP` [van der Pas 2017]. Assim, essa subseção avalia o desempenho da versão vetorizada proposta neste trabalho em conjunto com a paralelismo. O paralelismo do cálculo de interseção raio-triângulo é bem trivial, uma vez que o cálculo de cada par raio-triângulo pode ser realizado independente uns dos outros. Logo, a paralelização foi realizada através da inserção de uma diretiva `parallel for` do `OpenMP` antes do `for` (linha 3) da Figura 4.

Para avaliar o desempenho da versão `SIMD`+paralela foram utilizadas 3 instâncias denominadas de pequena, média e grande. Todas as instâncias são compostas de 16.777.216 raios. Por outro lado, as quantidades de triângulos adotadas para as instâncias pequena, média e grande, foram 2000, 10.000 e 50.000, respectivamente. A média aritmética do tempo de execução pode ser vista na Figura 5.

Observando o tempo de execução para as três instâncias, o desempenho melhora a medida que se aumenta a quantidade de *threads*. Os resultados obtidos são confiáveis com coeficiente de variação $< 4,5\%$. Uma melhor análise do desempenho de cada instância



(a) Pequeno (2K Triângulos) (b) Médio (10K Triângulos) (c) Grande (50K Triângulos)

Figura 5. Tempo de execução (segundos) da versão SIMD+Paralela

pode ser feita através dos $speedup = \frac{tempo(1)}{tempo(T)}$, onde T é o número de *threads*, apresentados na Figura 6. Apesar do *speedup* aumentar a medida que que se aumenta a quantidade de *threads* para todas as instâncias, quanto maior a granularidade da tarefa melhor o desempenho alcançado. Esse comportamento é esperado pois quanto menor a granularidade da tarefa maior a influência das sobrecargas de execução, ainda mais considerando a sobrecarga extra para se acessar a memória não local nas arquiteturas NUMA, que tem uma maior influencia nas tarefas de granularidade mais fina. Mesmo assim, o desempenho alcançado foi bastante consistente, especialmente para as instâncias média e grande.

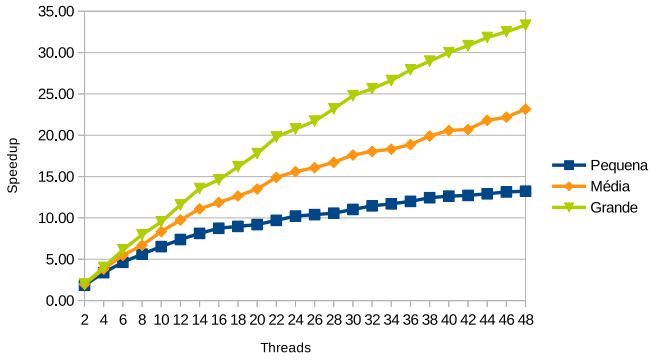


Figura 6. Speedup da versão SIMD+Paralelo para as três instâncias

Mais importante, quando comparamos o ganho de desempenho total (SIMD+Paralelismo), os valores alcançados são notáveis. Por exemplo, considerando o ganho máximo em relação a versão original ($speedup_{original} = \frac{tempo_{original(1)}}{tempo_{vetorizado(48)}}$), os *speedups* para as instâncias pequena, média e grande foram, respectivamente, $193,84$, $334,57$ e $369,88$. Por sua vez, se consideramos o ganho máximo em relação a versão otimizada os *speedups* foram $93,81$, $222,11$ e $257,42$. O que mostra que a vetorização proposta neste trabalho foi capaz de utilizar o potencial das arquiteturas *multicore*.

6. Conclusões

O cálculo de interseção raio-triângulo é a parte mais custosa da técnica de *Ray Tracing* que é muito utilizada na síntese de imagens. Por outro lado, para utilizar eficientemente as máquinas *multicore*, presentes em muitos laboratórios e centros de pesquisas, é necessário aproveitar tanto as instruções SIMD, assim como os múltiplos núcleos disponíveis nestas arquiteturas. Este trabalho propõe e implementa uma vetorização eficiente para utilizar o potencial das instruções SIMD. A abordagem proposta é escalável e, em conjunto com o

paralelismo inerente no cálculo de interseção raio-triângulo, consegue explorar o potencial das arquiteturas *multicore*, atingindo um desempenho até 257 vezes melhor do que a versão sequencial otimizada.

Agradecimentos

Os autores agradecem o uso dos recursos computacionais *multicore* mantidos e operados pelo Núcleo de Computação Científica da Universidade Estadual Paulista (NCC/UNESP), financiado parcialmente pela Intel, no contexto do projeto Intel/UNESP Modern Code.

Esta pesquisa é financiada pelos projetos CNPq 426729/2018-8 e FAPDF 00193-00002139/2018-79.

Referências

Diaz, J., Muñoz-Caro, C., and Niño, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386.

Glassner, A. S. (1989). *An introduction to ray tracing*. Elsevier.

Havel, J. and Herout, A. (2010). Yet faster ray-triangle intersection (using sse4). *IEEE Transactions on Visualization and Computer Graphics*, 16(3):434–438.

Mark-Sabahi (2012). A guide to auto-vectorization with intel c++ compilers. Technical report, Intel.

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 IEEE Latin American Symp. on Circuits and Systems*, pages 77–80.

Okmen, Y. (2011). SIMD Floating Point Extension for Ray Tracing. Master's thesis, Delft University, The Netherlands.

Sena, A. C., Nascimento, A., Vasconcelos, C., and Marzulo, L. A. J. (2017). Execução eficiente do algoritmo de leilão nas novas arquiteturas multicore. *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, 18(1/2017).

van der Pas, R., S. E. S. E. T. C. (2017). *Using OpenMP – The Next Step:Affinity, Accelerators, Tasking, and SIMD*. The MIT Press.

Wald, I., Benthin, C., and Boulos, S. (2008). Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 49–57.

Wende, F., Noack, M., Steinke, T., Klemm, M., Newburn, C. J., and Zitzlsberger, G. (2016). Portable simd performance with openmp* 4.x compiler directives. In *Proc. of the Inter. European Conference on Parallel Processing*, pages 264–277.

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

Woop, S., Brunvand, E., and Slusallek, P. (2006). Estimating performance of a ray-tracing asic design. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 7–14.