Utilizando a biblioteca PAPI para avaliar diferentes abordagens de construção de curvas b-spline

Vitor P. David[1], **Douglas Custodio de Araujo**[1], **Marcelo Zamith** [1], **Ubiratam de Paula** [1]

[1] Universidade Federal Rural do Rio de Janeiro (UFRRJ)

vitor.pito@gmail.com,custodiodouglas6@gmail.com

mzamith@ufrrj.br, upaula@ufrrj.br

Resumo. *Tendo em vista a importância da paralelização de trechos do algoritmo, para a obtenção de desempenho, este artigo propõe a análise de desempenho para diferentes níveis de abordagens paralelas permitindo concluir a melhor abordagem para curvas B-spline. O desempenho foi avaliado utilizando diferentes níveis de paralelização, por meio de instruções, com a vetorização, e por meio de núcleos, através das threads. Nossos resultados demonstraram que uma abordagem através de núcleos obteve um aproveitamento muito bom enquanto que a vetorização não obteve ganhos.*

Abstract. *Given the importance of parallelization of the algorithm sections to obtain performance, this paper proposes the performance analysis for different levels of parallel approaches allowing to conclude the best approach for b-spline curves. Performance was evaluated using different levels of parallelization, through instructions, with vectoring, and through cores, through threads. Our results demonstrated that a nucleus approach obtained very good performance while the vectorization did not.*

1. Introdução

Com o desenvolvimento do computador digital, a demanda por processamento vem crescendo. Assim, a estratégia para atender essa necessidade foi de buscar aumentar, de uma forma ou de outra, a vazão de instruções, seja pelo aumento da frequência do processador, seja pela implementação de memórias cache multinível.

Além disso, outras estratégias foram adotadas, como a tecnologia MMX(*Multimedia Extensions*), que foi incluída nos processadores Pentium, onde passou a fazer parte dos processares até os tempos atuais. É uma tecnologia amplamente empregada por aplicações multimídia [Peleg et al. 1997].

A estratégia de aumento da frequência do processador foi levada até os limites físicos dos materiais utilizados na construção do chip do processador [Stallings 2003]. Com a limitação imposta, outras estratégias foram desenvolvidas para aumentar a vazão de instruções e, consequentemente, reduzindo o tempo de computação de uma tarefa.

Processadores escalares e simétricos, bem como *super pipelines* foram algumas as melhorias em hardware para reduzir o tempo de processamento. Atualmente, os processadores *multicores* equipam tanto os computadores pessoais quanto os dispositivos móveis, o que permite ao S.O (Sistema Operacional) executar diferentes tarefas simultaneamente.

Além disso, a evolução dos processadores também incluiu registradores de fluxo com 256 e 512 bits, viabilizando uma vazão maior de dados processados. Equipando tanto processadores de custo baixo, utilizados em computadores pessoais, quanto os processadores empregados nos grandes *clusters* computacionais. Sendo os registradores vetorizados uma realidade comum em quase todos os computadores.

A cada nova tecnologia embarcada nos processadores, um conjunto de instruções é necessário para ativá-la. Nesse sentido, existem instruções de vetorização disponíveis em bibliotecas que podem ser utilizadas de forma direta, como as instruções *intrinsics*[1] e a vetorização feita de forma automática pelo compilador.

A vetorização realizada com base nas instruções *intrinsics* apresenta um resultado melhor do que a forma automática. Porque ao escrever o código utilizando tais instruções, o desenvolvedor precisa modelar seus dados na forma SIMD(*Single Instruction Multiple Data*). Além disso, existem problemas computacionais que são naturalmente SIMD, exigindo apenas a modelagem certa dos dados para utilizar a tecnologia MMX. Por outro lado, a vetorização de forma automática fica a cargo do compilador, que nem sempre detecta que os dados podem ser estruturados na forma SIMD.

Para avaliar a performance do computador em relação ao paralelismo disponível nos processadores atuais, é proposto neste trabalho três abordagens de construção de curvas *b-spline*. Uma abordagem utilizando instruções *intrinsics*; uma outra abordagem baseada em *threads*, utilizando a biblioteca *pthread*; e, uma abordagem executando as instruções *intrinsics* em diferentes *threads*.

As curvas *b-spline* são amplamente utilizadas na computação gráfica para geração de curvas e superfícies suaves e apresentam controle local do formato da curva [Buss 2003], com especial atenção para os jogos digitais, onde o tempo de processamento é fundamental para garantir a interatividade do jogo.

Assim sendo, resultados obtidos neste trabalho serão utilizados tanto para avaliar a performance do processador quanto para avaliar a construção das curvas *b-spline* em aplicações de tempo real, como jogos digitais.

Para isso, é necessário modelar a construção das curvas no formato SIMD a fim de avaliar a performance dos registradores MMX. A abordagem baseada em *threads* utiliza o Algoritmo sequencial aplicado em conjuntos de dados; e, a abordagem baseada em *threads* com os registradores MMX processa o Algoritmo baseado em instruções MMX em cada *thread*.

Este trabalho utiliza a biblioteca API PAPI [Garner et al. 2000] para coletar as métricas e eventos necessários para avaliação da performance da aplicação junto ao hardware utilizado.

A literatura de implementações de algoritmos vetorizados é extremamente grande, portanto, iremos citar apenas alguns estudos os quais consideramos próximos ao assunto abordado neste estudo.

Em Stock et al. [Kevin Stock 2012], os autores discutiram uma solução para encontrar a melhor forma de vetorizar um algoritmo, utilizando técnicas de inteligência

[1] `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#`

artificial. O artigo apresenta um ganho de desempenho relevante, de 2 a 8 vezes, em relação aos melhores compiladores da época.

Bramas, B. [Bramas 2017] propôs a implementação vetorizada de um algoritmo clássico conhecido na literatura, Quick Sort. Nesse trabalho, foram utilizadas instruções AVX-512 considerando os processadores Intel Skylake. Esse trabalho mostra que algoritmos antigos podem ser melhorados para se obter um ótimo desempenho em arquiteturas atuais.

2. Abordagem paralela e vetorizada para geração de curvas B-Spline cúbica

A curva *B-Spline* faz parte de uma família de curvas conhecidas como *Splines* [Buss 2003]. Uma das principais características deste tipo de curva é o uso de pontos de controle, que são usados para ajustar apenas uma seção da curva. Além disso, a curva gerada é aproximada em relação aos pontos de controle.

Assim sendo, a curva *B-Spline* uniforme é definida por uma sequência de pontos de controle: p_0, p_2, p_3, ..., p_n, que junto a um conjunto de funções: $f_0(u)$, $f_1(u)$, $f_2(u)$, $f_3(u)$, ..., $f_n(u)$ definem um polinômio cúbico que descreve a curva, conforme a Equação:

$$q(x) = \sum_{i=0}^{3} f_i(x) \times p_i \tag{1}$$

Os pontos de controle são aproximados e, portanto, o polinômio dado por $q(x)$ aproxima-se dos pontos p_i. Curvas baseadas em *B-spline* cúbicas podem ser divididas em seções, onde cada seção é definida por um conjunto de quatro pontos consecutivos, i.e., os pontos de controle [Wolberg 1990, Buss 2003]. A Figura 1 ilustra a influência dos pontos de controle sobre cada seção da curva.

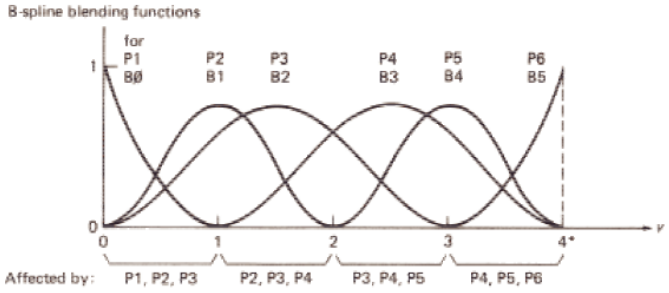


Figura 1. Exemplo de curvas e a influência dos pontos de controle [Wolberg 1990].

As funções $f_i(x)$ podem ser definidas segundo suas propriedades [Wolberg 1990, Buss 2003], conforme o conjunto de Equações (2) e podem ser reescritas na forma matricial (3):

$$
\begin{aligned}
f_0(x) &= \frac{1}{6}(1-x)^3 \\
f_1(x) &= \frac{1}{6}(3x^3 - 6x^2 + 4) \\
f_2(x) &= \frac{1}{6}(-3x^3 + 3x^2 + 3x + 1) \\
f_3(x) &= \frac{1}{6}(x^3)
\end{aligned}
\tag{2}
$$

$$
M = \frac{1}{6}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 4 \\ -3 & 3 & 3 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{3}
$$

Os pontos de controle podem ser escritos na forma vetorial: $P = (x_i, x_{i+1}, x_{i+2}, x_{i+3})$ e um vetor $H = (t^3 + t^2 + t + 1)$. Logo, o Algoritmo 1 descreve a construção da curva *B-spline*, considerando sua forma matricial.

Algoritmo 1: Algoritmo sequencial para obter a curva aproximada por *B-spline*.

Entrada: Matriz M, vetor H, conjunto de pontos de controle P, N
Saída: Curva B-spline

1 **início**
2 **para** $i + 3 < N$ **faça**
3 **para** $(t = 0, t \leq 1, t = t + h)$ **faça**
4 $\bar{x} \leftarrow \frac{(H \cdot M)}{6} \begin{bmatrix} x_i \\ x_{i+1} \\ x_{i+2} \\ x_{i+3} \end{bmatrix}$
5 $\bar{y} \leftarrow \frac{(H \cdot M)}{6} \begin{bmatrix} y_i \\ y_{i+1} \\ y_{i+2} \\ y_{i+3} \end{bmatrix}$
6 **fim**
7 $i \leftarrow i + 1$
8 **fim**
9 **fim**

A linha 2 do Algoritmo (1) garante que são utilizados quatro pontos consecutivos. A partir desses pontos de controle, a curva é construída e é calculado o polinômio que vai encontrar os valores de $\bar{x}$ e $\bar{y}$, respectivamente nas linhas 4 e 5 do Algoritmo (1).

Para gerar uma curva longa capaz de representar diversas formas, é necessário utilizar centenas de milhares de pontos de controle, onde cada conjunto de quatro pontos consecutivos controla uma seção local da curva. Nesse caso, o tempo de computação para a geração de toda a curva exigirá um tempo maior de computação.

De fato, o tempo computacional é diretamente relacionado ao tamanho do conjunto de pontos de controle. É possível ainda afirmar que o Algoritmo 1 apresenta complexidade $O(n)$, onde n é a quantidade de pontos de controle.

O trabalho adota três estratégias de paralelismos do Algoritmo 1: i) o paralelismo a nível de instruções, utilizando a biblioteca *intrinsics*; ii) a abordagem baseada em *threads*, onde Algoritmo 1 é executando por diferentes *threads* e cada *thread* fica responsável por gerar uma seção da curva, para isso é utilizada a biblioteca *pthread*; e, iii) onde a abordagem paralela busca combinar o paralelismo a nível de instruções em diferentes *threads*, para isso, foi combinada as implementações utilizando a biblioteca *intrinsics* com a biblioteca *pthread*.

Estratégia baseada na biblioteca *intrinsics*: Esta abordagem realiza a multiplicação das matrizes de forma vetorizada, ou seja, as linhas e colunas são multi-

plicadas de quatro em quatro elementos, que é a quantidade de números de dupla precisão(cada um com 64 bits totalizando 256 bits) que cabem no registrador vetorizado.

Desse modo, foram realizadas duas abordagens: a primeira é a de vetorizar a multiplicação $A = \frac{H*M}{6}$ e depois a vetorização $A * P$; a segunda abordagem realiza a multiplicação $A = \frac{H*M}{6}$ com instruções sequenciais e a multiplicação $A * P$ de forma vetorizada.

Estratégia baseada na biblioteca *pthread*: divide o vetor P entre as *threads*, dessa maneira cada *thread* executa $\frac{N-3}{T}$ vezes a operação $\frac{H*M*P}{6}$, sendo T o número de *threads*.

Estratégia baseada em paralelismo de instruções executada por diversas *threads*: essa abordagem divide o vetor P entre as *threads* e cada *thread* executa a primeira abordagem vetorizada, onde a multiplicação $A = \frac{H*M}{6}$ é feita de forma sequencial e o produto $A * P$ é vetorizado.

3. Testes realizados

Para realizar os testes foi utilizada uma máquina com um processador i7 com 4 núcleos físicos e 8 núcleos com *HyperThread* com 4GHz de frequência tendo uma cache L3 de 3MB, duas caches L2 de 250KB e quatro caches L1 de 32KB cada, a cache L3 é divida para todos os núcleos, as L2 são divididas entre dois núcleos e cada núcleo possui sua própria cache L1. A máquina também possui 16GB de memória principal, sistema operacional Ubuntu 18.04 e gcc na versão 8.3.0 com parâmetro $O1$ de otimização.

A alocação da memória foi alinhada com a linha de cache em 64 bytes em todas as implementações. Os autores desenvolveram diferentes implementações do Algoritmo 1, de acordo com a lista:

- **b-spline**: Algoritmo *b-spline* sem utilização da função pow e sequencial, abordagem utilizada como referência para o cálculo da aceleração.
- **b-spline-vet**: Versão vetorizada para construção da curva.
- **b-spline-p2**: Versão paralela com duas *threads* para a construção da curva, sem vetorização.
- **b-spline-p4**: Versão paralela com quatro *threads* para a construção da curva, sem vetorização.
- **b-spline-p8**: Versão paralela com oito *threads* para a construção da curva, sem vetorização.
- **b-spline-p2-vet**: Versão paralela com duas *threads* para a construção da curva, com vetorização.

Para avaliar os resultados, foi utilizada a biblioteca PAPI, onde foram coletadas as seguintes métricas:

- L1_DCM: Quantidade de falha de cache na cache L1 de acesso a dados
- L2_DCM: Quantidade de falha de cache na cache L2 de acesso a dados
- L3_DCM: Quantidade de falha de cache na cache L3 de acesso a dados
- BR_MSP: Número de instruções de decisão com falha de predição
- RES_STL: Tempo de processamento pausado devido a espera por recursos, em ciclos

Para cada teste, foi obtida a média de 10 execuções e os dados variaram de tamanho de 1024 Bytes até 256MBytes. Os testes consideram um conjunto de pontos de controle baseados em um *float* de 32bits. Logo, considerando as duas dimensões do problema, então com 1024 Bytes tem-se um vetor com 128 pontos e com 256MBytes o vetor tem 33.554.432 pontos de controle.

Antes de cada execução foi alocada e executada uma computação sobre um vetor de inteiros de 20MBytes com o objetivo de garantir que as caches estivessem sem qualquer dado do problema, interferindo de alguma forma na coleta das métricas.

A Figura (2) apresenta o tempo em segundos para cada instância. O melhor resultado foi obtido com a abordagem de 8 *threads*, que alcançou um *speedup* máximo de $4,11$ vezes em relação ao sequencial para 128 e 256 mega bytes, conforme ilustra a Tabela (1).

É possível observar que para problemas muito pequenos, aproximadamente entre 1Kbyte e 32Kbytes, a abordagem sequencial é melhor. A partir de 64Kbytes, as versões paralelas passam a ganhar no tempo de computação.

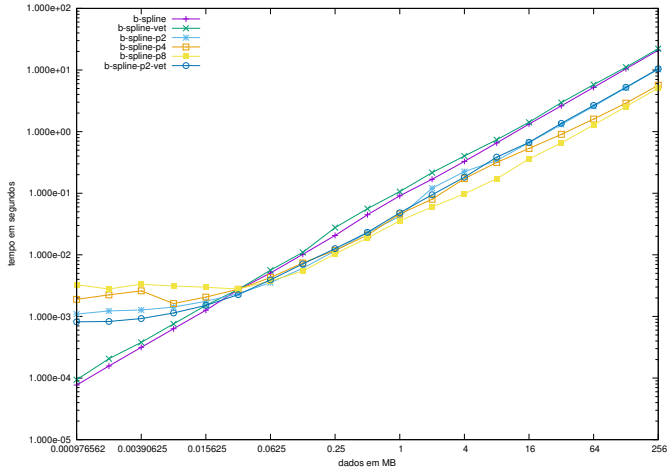


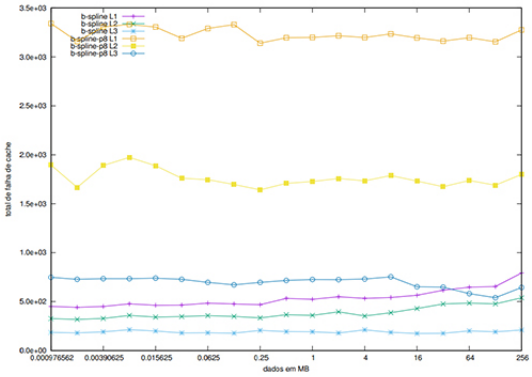
Figura 2. Tempo de computação das instâncias em segundos.

As Figuras 3(a) e 3(b) mostram a comparação entre a implementação mais rápida, a abordagem paralela com 8 *threads*, e a abordagem mais lenta, sequencial.

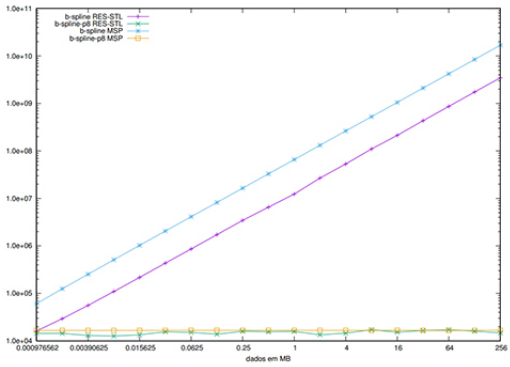
A quantidade total de falhas de cache na abordagem paralela é maior que na versão sequencial, conforme ilustra a Figura 3(a). Esse resultado é coerente, uma vez que as *threads* competem pelo uso das caches. Considera-se que cada *thread* computa um seção da curva e, para isso, cada *thread* recebe o seu conjunto de pontos consecutivos.

É importante notar que mesmo não utilizando as memórias cache de forma eficiente, a abordagem paralela é mais rápida. Logo, a questão é entender a origem desse ganho.

Nesse sentido, a Figura 3(b) mostra outras duas métricas as quais avaliam de forma implícita o *pipeline*. A métrica PAPI_RES_STL fornece a informação de quanto tempo o programa ficou aguardando um recurso ocupado, enquanto que a métrica PAPI_BR_MSP fornece a contagem de vezes em que houve uma predição errada em algum desvio condicional.



(a) Total de falhas de cache.



(b) Falha de predição de desvios.

Figura 3. Memória cache e *pipeline*.

Dados	Tempo seq.	Tempo par.	*speedup*
1024	0.0077	0.3269	0.0236
2048	0.0157	0.2794	0.0562
4096	0.0315	0.3341	0.0943
8192	0.0632	0.3108	0.2033
16384	0.1261	0.2982	0.4229
32768	0.2715	0.2781	0.9763
65536	0.5078	0.3738	1.3585
131072	1.0188	0.5463	1.8649
262144	2.0656	1.0365	1.9929
524288	4.4736	1.8754	2.3854
1048576	9.0961	3.5636	2.5525
2097152	16.8749	5.9723	2.8255
4194304	32.7697	9.7812	3.3503
8388608	65.3852	17.2281	3.7953
16777216	132.6137	36.0122	3.6825
33554432	262.1257	65.5347	3.9998
67108864	522.3981	128.5494	4.0638
134217728	1,046.6290	254.0821	4.1193
268435456	2,096.8820	509.8331	4.1129

Tabela 1. Tempo sequencial $\times$ paralelo.

Ainda sobre a Figura 3(b), também é possível analisar o comportamento assintótico exponencial em função da quantidade de dados trafegados pelas memórias para a abordagem sequencial. Dessa forma, percebe-se que a abordagem paralela com *threads* aproveita melhor o *pipeline*, sendo essa a principal razão dos ganhos alcançados com a abordagem paralela.

A abordagem com 8 *threads* utiliza todo o recurso de *HyperThread* trazida pelo processador, o qual conta com quatro núcleos físicos e 8 em *HyperThread* e, portanto, apesar de apresentar alguma aceleração em relação o sequencial, o ganho não é de oito vezes o sequencial.

O melhor aproveitamento, realizado pela abordagem paralela com 8 *threads*, faz com que essa abordagem apresente um ganho expressivo, ainda que não aproveite de forma eficiente as caches.

Em relação a ao uso da memória cache, as falhas de cache nas abordagens vetorizadas foi maior que na implementação sem vetorização. E, nas paralelas foram mais significativas. O número de falhas de cache não sofreu influência de acordo com o tamanho do vetor, porém pode-se perceber que quanto maior o número de *threads* maior foi o número de falhas na L1 e L2, não sendo observado este comportamento na cache L3.

A quantidade de falhas de predição não mostrou diferença significativa com relação a abordagem vetorizadas e não vetorizada. Por outro lado, a estratégia baseada em *threads* ativou uma grande quantidade de falhas de predição, que não teve qualquer

relação com a quantidade de *threads* alocadas. Por fim, em todas as estratégias foi observada uma baixa quantidade de espera de recursos, mesmo na abordagem *multithreads*.

Os testes mostraram que o tipo de aplicação não é ideal para ser modelada na forma SIMD usando as instruções *intrinsics*. Além disso, a abordagem que obteve melhor resultado foi baseada em *threads*, limitando o números de *threads* ao número de cores físicos do computador.

4. Conclusão e trabalhos futuros

É notável que abordagens paralelas para o processamento de algoritmos torna a execução consideravelmente mais rápida. Das estratégias apresentadas neste trabalho, a abordagem baseada em *threads* obteve um ganho significativo e os testes demonstraram que utilizar um número maior de *threads* que o de núcleos é, entretanto, ineficiente.

Embora a vetorização tenha obtido um desempenho ruim para esse tipo de aplicação, sendo até pior que a execução da abordagem sequencial, este trabalho tornou possível a comparação das diferentes abordagens, buscando a abordagem ideal para este problema.

Além do tempo de execução, outras métricas foram coletadas utilizando a biblioteca PAPI afim de obter alguns outros resultados sobre a execução da aplicação. Assim sendo, pode-se observar como as caches e o pipeline se comportaram, especialmente para problemas que ocupam mais que 64Kbytes.

Como trabalhos futuros pretende-se estudar mais as aplicações vetorizadas e paralelas para obter uma melhor utilização da cache e dos recursos, visando melhorar ainda mais os algoritmos paralelos e obter um melhoria nos vetorizados.

Referências

Bramas, B. (2017). A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake. *International Journal of Advanced Computer Science and Applications(IJACSA)*, 8.

Buss, S. R. (2003). *3D computer graphics: a mathematical introduction with OpenGL*. Cambridge University Press.

Garner, B. D., Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P. (2000). A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204.

Kevin Stock, Louis-Noël Pouchet, P. S. (2012). Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO) - Special Issue on High-Performance Embedded Architectures and Compilers*, 8(50).

Peleg, A., Wilkie, S., and Weiser, U. (1997). Intel mmx for multimedia pcs. *Communications of the ACM*, 40(1):24–38.

Stallings, W. (2003). *Computer organization and architecture: designing for performance*. Pearson Education India.

Wolberg, G. (1990). *Digital image warping*, volume 10662. IEEE computer society press Los Alamitos, CA.