Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação
Desenvolvimento de Código Otimizado - SSC 0951

Relatório Trabalho 4

Vetorização

1. Lucas G. Meneses Número: 13671615
2. Henrique S. Marques Número: 11815722
3. Carlos F. C. Lemos Número: 12542630

28 de novembro de 2023,
São Carlos, SP,
Brasil

Conteúdo

1 Introdução

A vetorização de código é uma técnica crucial na otimização de desempenho de programas, especialmente em computação de alto desempenho e programação paralela. Essa abordagem visa aproveitar as capacidades dos processadores modernos, que muitas vezes são equipados com conjuntos de instruções SIMD (*Single Instruction, Multiple Data*). Essas instruções permitem que operações sejam executadas, simultaneamente, em vetores ou matrizes, acelerando significativamente o processamento. Existem duas abordagens principais para vetorização de código: a vetorização automática via compilador e a vetorização manual pelo uso de instruções específicas (*intrinsics*).

Ao compilar código-fonte, os compiladores modernos, como o `g++`, podem analisar *loops* e operações em *arrays* e automaticamente transformar o código para aproveitar instruções SIMD quando possível. A opção de compilação `-ftree-vectorize` no `g++` ativa a vetorização automática, permitindo ao compilador decidir quais partes do código podem ser vetorizadas. Por outro lado, a vetorização manual envolve o programador utilizar funções fornecidas por bibliotecas de desenvolvimento para representar diretamente as instruções SIMD. Essas instruções permitem ao programador ter controle total sobre a vetorização, escolhendo as operações específicas e otimizando para casos particulares.

Neste contexto, a tarefa proposta visa comparar a eficácia da vetorização

automática pelo compilador, a vetorização manual usando instruções *intrinsics* e a não-vetorização, no que se diz respeito ao acesso à memória *cache* e tempo de execução.

2 Metodologia

Para a realização de todos os experimentos, foi utilizado um computador rodando o Sistema Operacional Ubuntu, cuja CPU é um Intel Core i5 6600k com frequência de *clock* de 3,5 GHz, 4 núcleos e 4 *threads*. Quanto à memória, o computador faz uso de 16 GB de RAM do tipo DDR4, cache L1 de dados e de instruções ambas 8-*way* com 4×32 kB de capacidade, L2 4-*way* 4×256 kB e L3 12-*way* de 6 MB.

Nesse sentido, a não-vetorização de código, a vetorização automática via compilador (`g++`) e a vetorização por meio de instruções *intrinsics* foram testadas através de um código contendo multiplicação e soma de matrizes 24×24. Desse modo, cada um dos testes foi executado 10 vezes, sendo cálculado as médias aritméticas e intervalos de confiança (95 % das métricas: (i) tempo de execução (Seção 3.1); (ii) quantidade total de acessos à memória *cache* (Seção 3.2); (iii) quantidade total de *cache misses* (Seção 3.3). Por fim, o código e os dados obtidos estão disponíveis em repositório aberto via GitHub.

3 Resultados

3.1 Tempo de Execução

Os resultados dos experimentos apresentados na Tabela 1 revelam claramente o impacto significativo da vetorização no desempenho do código. Ao comparar os tempos de execução, observamos que a versão sem vetorização possui tempos consideravelmente mais elevados, com uma média aritmética de aproximadamente 3,1 s. Em contraste, a vetorização pelo compilador reduz drasticamente os tempos para uma média de 0,002 s, enquanto a vetorização intrínseca apresenta um desempenho ligeiramente superior, com uma média de 0,004 s.

Ao analisar os intervalos de confiança a 95 % de confiança, observamos que as variações nos tempos de execução são mais consistentes na vetorização pelo

compilador, indicando uma estabilidade maior em comparação com a vetorização pelo programador. Em última análise, esses resultados destacam a importância da vetorização no contexto de otimização de código, permitindo uma execução mais eficiente e rápida de operações em matrizes.

Tabela 1: Tempos de execução (s), médias aritméticas e intervalos de confiança obtidos para os 10 experimentos realizados para o código sem vetorização, com vetorização pelo compilador e com vetorização *intrinsics*

Experimento	**Sem Vetorização**	**Vetorização Compilador**	**Vetorização *Intrinsics***
Tempos de Execução (s)	3,071371827	0,001703616	0,006194783
	3,086012535	0,001876712	0,003267244
	3,066365197	0,001666195	0,003275400
	3,083345884	0,002079098	0,006265360
	3,076254198	0,001561162	0,003407487
	3,086721908	0,001814857	0,006235915
	3,137937281	0,001705785	0,003359132
	3,075123578	0,002215630	0,003373285
	3,081899078	0,001904354	0,003202259
	3,258685771	0,001696352	0,003296298
Média Aritmética (s)	3,1023717	0,0018224	0,0041877
Intervalo de Confiança (95 %)	[3,0606087 , 3,1441347]	[0,0016780 , 0,0019667]	[0,0031776 , 0.0051978]

3.2 Acesso à *Cache*

Os resultados relacionados ao acesso à memória *cache*, apresentados na Tabela 2, oferecem alguns *insights* sobre o impacto da vetorização nas operações de leitura e gravação na *cache*. A análise dos números totais de acessos à *cache* revela uma diferença notável entre os diferentes métodos de otimização. Na versão sem vetorização, o número médio de acessos é significativamente maior, atingindo uma média aritmética de $407954,9$ acessos. Em contraste, a vetorização pelo compilador e a vetorização intrínseca resultam em reduções substanciais, com médias de $142426,0$ e $140586,1$ acessos, respectivamente.

Essa discrepância nos números de acessos à *cache* pode ser explicada pela capacidade da vetorização de processar dados em paralelo, reduzindo a necessidade de acessos frequentes à memória. A vetorização, portanto, otimiza o uso da *cache*, minimizando a movimentação de dados e atuando na melhoria de eficiência do código, conforme discutido na Seção 3.1.

Ao analisar os intervalos de 95 % de confiança, observamos que, embora a média de acessos seja menor nas versões vetorizadas, a variação nos resultados é relativamente consistente em ambas as abordagens de vetorização. Isso sugere que, em termos de acessos à *cache*, tanto a vetorização pelo compilador quanto a vetorização manual oferecem estabilidade similar.

Tabela 2: Número total de acessos à memória *cache*, médias aritméticas e intervalos de confiança obtidos para os 10 experimentos realizados para o código sem vetorização, com vetorização pelo compilador e com vetorização *intrinsics*

Experimento	**Sem Vetorização**	**Vetorização Compilador**	**Vetorização *Intrinsics***
Número de Acessos à *Cache*	203088	146013	147808
	514254	148009	138266
	177207	147162	137225
	560863	137725	141209
	227324	137395	134835
	518490	141583	142374
	505803	141503	149419
	473696	139839	140076
	519125	146161	138004
	379699	138870	136645
Média Aritmética	407954,9	142426,0	140586,1
Intervalo de Confiança (95 %)	[300838,5 , 515071,3]	[139518,0 , 145334,0]	[137163,9 , 144008,3]

3.3 *Cache Misses*

Ao comparar os resultados descritos na Seção 3.2 e a Tabela 3, fica evidente a relação entre a redução do número total de acessos à *cache* e a diminuição dos *cache misses*. Os *cache misses* ocorrem quando o processador não consegue encontrar os dados necessários na memória *cache*, sendo forçado a buscar na memória principal. Portanto, ao reduzir o número geral de acessos à *cache*, a vetorização contribui diretamente para a mitigação dos *cache misses*, resultando em operações de memória mais eficientes e rápidas. Os intervalos de confiança a 95 % reforçam a consistência das versões vetorizadas, indicando uma variação menor nos resultados.

Tabela 3: Número total de *cache misses*, médias aritméticas e intervalos de confiança obtidos para os 10 experimentos realizados para o código sem vetorização, com vetorização pelo compilador e com vetorização *intrinsics*

Experimento	**Sem Vetorização**	**Vetorização Compilador**	**Vetorização *Intrinsics***
Número de *Cache Misses*	64226	49934	67931
	84730	48494	51972
	59253	48652	45000
	58296	45505	51132
	72746	45474	66533
	87831	47280	51901
	68527	45962	52558
	60152	45964	64717
	57575	49447	45762
	230276	45721	51718
Média Aritmética	84361,2	47243,3	54922,4
Intervalo de Confiança (95 %)	[46879,59 , 121842,81]	[45995,26 , 48491,34]	[48935,44 , 60909,36]

4 Conclusões

A análise dos resultados obtidos, referentes aos tempos de execução, acessos à *cache* e *cache misses*, proporcionou alguns *insights* sobre a influência direta da vetorização no desempenho do código. A partir dos dados apresentados na Tabela 1, tornou-se evidente que a vetorização, tanto pelo compilador quanto pelo programador, desempenha um papel crucial na redução significativa dos tempos de execução. A capacidade de realizar operações em paralelo em conjuntos de dados permitiu uma execução mais eficiente, culminando em tempos de execução notavelmente inferiores em comparação com a versão não vetorizada.

Ao abordar os aspectos de acesso à memória *cache*, conforme evidenciado nas Tabelas 2 e 3, observou-se uma redução significativa no número total de referências à *cache*, contribuindo diretamente para a minimização de *cache misses*. A vetorização emergiu, portanto, como uma estratégia eficaz para otimizar o comportamento global de acesso à memória e, consequentemente, o desempenho computacional como um todo. A consistência nos resultados das versões vetorizadas, evidenciada pelos intervalos de confiança a 95 %, reforça a robustez dessas otimizações.