

<b>SSC0951</b>	<b>Desenvolvimento de Código Otimizado</b>
<b>Atividade 5 e 6</b>	<b>Vetorização de Código e Técnica de Blocagem</b>
<b>Nome:</b>	<b>Gabriel Van Loon Bodê da Costa Dourado Fuentes Rojas</b> <b>Alberto Campos Neves</b>

---

## 1. Introdução

Nesta prática iremos analisar o efeito da vetorização de código em um programa escrito na linguagem C e compilado utilizando o compilador GCC.

Em resumo, a atividade está dividida em 5 partes tais que, na parte 2 discorreremos acerca do planejamento utilizado neste relatório e os fatores e métricas considerados durante os experimentos, na parte 3 descrevemos o código utilizado durante o experimento e as versões implementadas, na parte 4 relatamos os resultados obtidos por meio da execução e análise dos dados e, por fim, na parte 5 discutimos as conclusões inferidas dos resultados obtidos na atividade e uma melhoria aplicada após a análise dos resultados obtidos (*Referenciada como Caso Extra nos resultados*).

- **Compilador:** g++ (Ubuntu 9.3.0-10ubuntu2) 9.3.0
- **Arquitetura:** intel x86\_64 (Intel(R) Core(TM) i5 CPU 650 @ 3.20GHz)
- **Sistema Operacional:** Ubuntu
- **Versão do Kernel:** Linux 5.4.0-33-generic
- **Tamanho da L1-dcache:** 64 KiB

## 2. Planejamento de Experimento

Neste experimento não utilizamos nenhuma das técnicas vistas anteriormente por estarmos variando um único fator: a técnica de vetorização de código. Os diferentes níveis sendo comparados são os seguintes:

- ❑ **Caso base:** neste caso o programa é compilado com a versão *naive* da implementação e sem o uso de nenhuma flag de otimização.
- ❑ **Vetorização GCC:** neste caso pedimos para que o GCC compile nosso código automaticamente ao incluir a flag *-ftree-vectorize* junto ao comando de compilação.
- ❑ **Vetorização Intrínseca:** neste caso reescrevemos manualmente o código para uma versão otimizada utilizando-se das funções e registradores definidos na arquitetura intel em sua extensão **SSE3**.
- ❑ **SSE3 + Blocagem:** este caso não constava no planejamento inicial e foi criado após a análise dos resultados anteriores. Consistem em uma otimização da função do experimento anterior com a **técnica de blocagem** (mais detalhes no capítulo 5).

Em cada um dos experimentos foi utilizado a ferramenta *perf* inclusa no toolkit do linux para avaliar as seguintes métricas: *tempo de execução*, *cache-references*, *cache-misses*, *branches* e *branch-misses*.

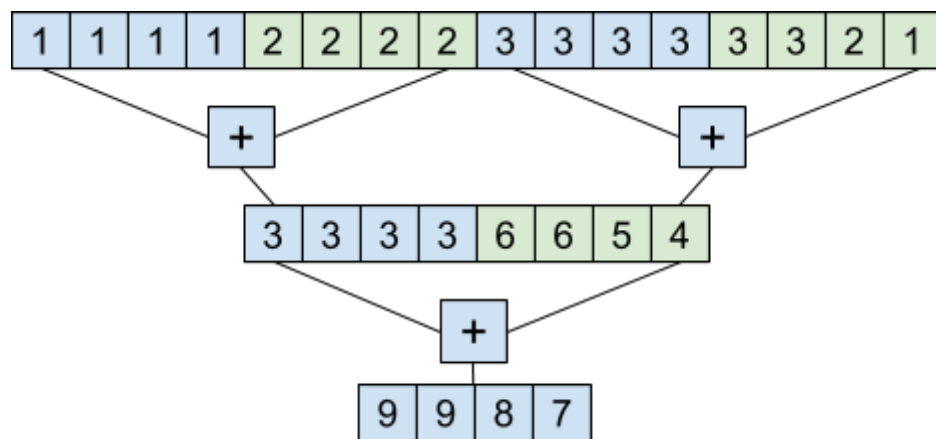
### 3. Descrição do Programa

O programa utilizado como objeto para análise dos 3 experimentos se trata de um programa simples escrito na linguagem C que implementa uma função que soma todos os elementos em um vetor e retorna o valor total.

```
float sum_default(float* v){  
    int i;  
    float sum = 0;  
    for(i = 0; i < N; i++){  
        sum += v[i];  
    }  
    return sum;  
}
```

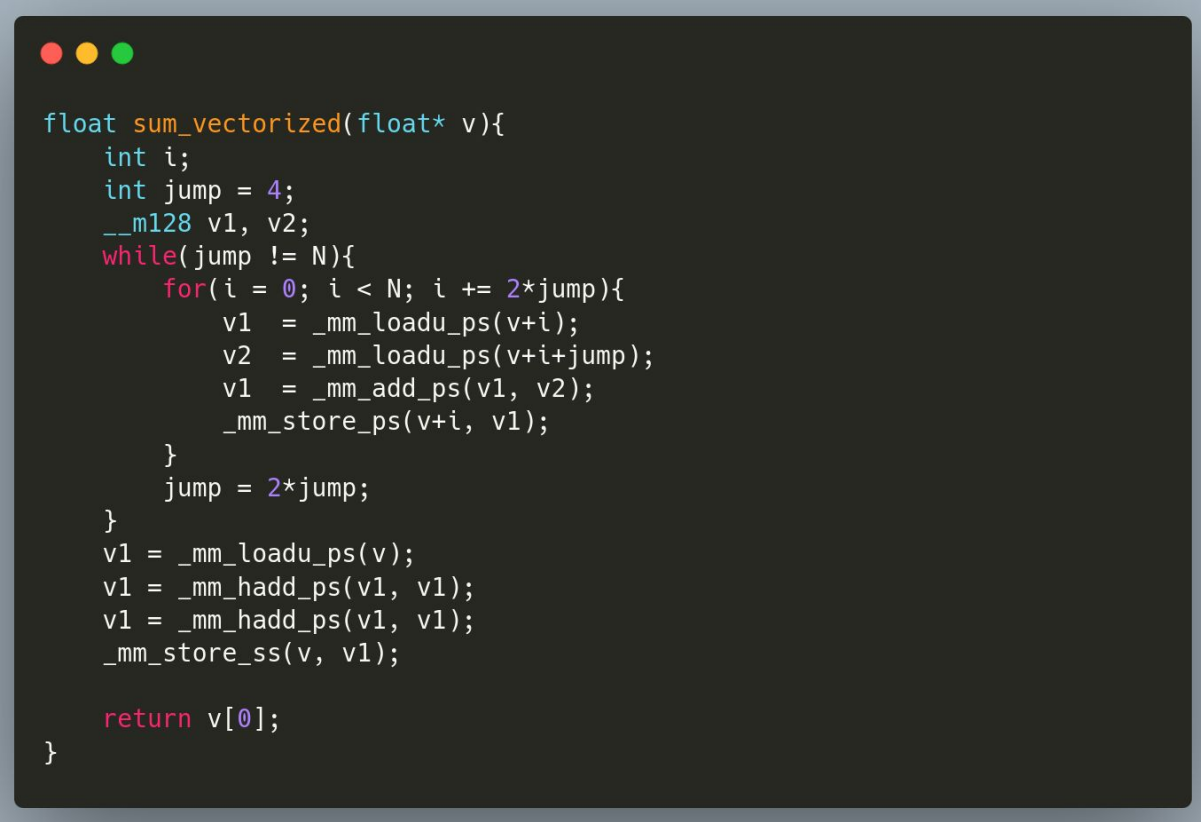
**Imagem 1** - versão naive do código que soma os elementos e retorna o valor total.

Dentre as possíveis formas de vetorização do código acima, decidimos pela abordagem de somar o vetor o tratando como uma árvore invertida, de tal forma que as somas pudessem ser feitas de 4 em 4 elementos utilizando os vetores definidos pela SSE3 e, no final, o resultado ficasse armazenado na primeira posição do vetor.



**Imagem 2** - esquemático da soma implementada em SSE3

O resultado final da função implementada é apresentado a seguir (imagem 3) e vemos que foi necessário aumentar um nível de aninhamento para permitir que a função fosse implementada corretamente de maneira iterativa.

A screenshot of a code editor with a dark background and light-colored text. The code is written in C and implements a vectorized sum function. It uses SSE intrinsics for loading, adding, and storing data. The function has a while loop that iterates over the array, and a for loop inside that processes elements in chunks of 4. The code is as follows:

```
float sum_vectorized(float* v){
    int i;
    int jump = 4;
    __m128 v1, v2;
    while(jump != N){
        for(i = 0; i < N; i += 2*jump){
            v1 = _mm_loadu_ps(v+i);
            v2 = _mm_loadu_ps(v+i+jump);
            v1 = _mm_add_ps(v1, v2);
            _mm_store_ps(v+i, v1);
        }
        jump = 2*jump;
    }
    v1 = _mm_loadu_ps(v);
    v1 = _mm_hadd_ps(v1, v1);
    v1 = _mm_hadd_ps(v1, v1);
    _mm_store_ss(v, v1);

    return v[0];
}
```

**Imagem 3** - versão vetorizada do código que soma os elementos e retorna o valor total.

## 4. Resultados Obtidos

Na tabela abaixo exibimos os resultados obtidos com o caso base e em seguida com as outras duas técnicas de vetorização empregadas no código e na compilação do projeto.

Experimento	branches	branch-misses	branch-misses (%)	Tempo Exec.
Caso Base	5114437	16635	0,0032	0.010992
Vetorização GCC	5119972	16991	0,0033	0.011043
Vetorização Intrínseca	3549536	16862	0,0047	0.012817
SSE3 + Blocagem	3462750	15535	0,0045	0.009452
Experimento	cache-refs	cache-misses	cache-misses (%)	Tempo Exec.
Caso Base	101801	82089	0,8063	0.011002
Vetorização GCC	99861	80868	0,8098	0.010528
Vetorização Intrínseca	160811	121536	0,7557	0.012393
SSE3 + Blocagem	100057	80887	0,8084	0.008986

**Tabela 1:** Média dos resultados obtido em cada métrica nos experimentos realizados.

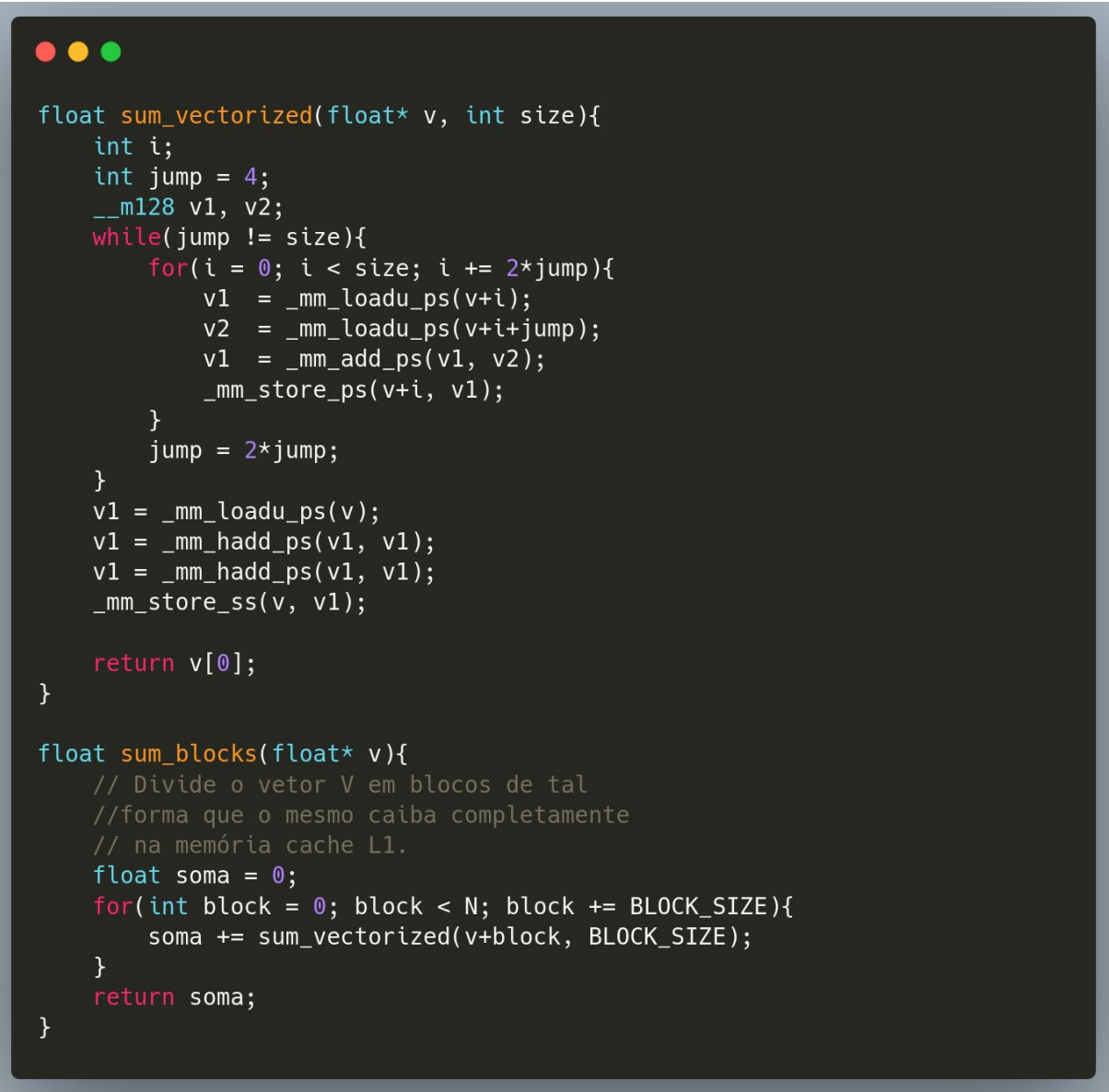
A seguir colocamos também uma tabela contendo a variação percentual das duas técnicas utilizadas em relação ao caso base.

Experimento	branches	branch-misses	Tempo Exec.
Caso Base	0.000000	0.000000	0.000000
Vetorização GCC	-0,001082	-0,021401	-0,004640
Vetorização Intrínseca	0,305977	-0,013646	-0,166030
Caso Extra (*)	0.322946	0.066125	0.140101
Experimento	cache-refs	cache-misses	Tempo Exec.
Caso Base	0.000000	0.000000	0.000000
Vetorização GCC	0.019056	0.014874	0.043083
Vetorização Intrínseca	-0.579660	-0.480539	-0.126431
Caso Extra (*)	0.017131	0.014642	0.183239

**Tabela 2:** Variação percentual dos experimentos em relação ao caso base. Cores acentuadas baseado na intensidade da variação.

## 5. Conclusões

Após o término dos experimentos, reparamos que o método empregado por nós para vetorizar a função de soma conseguiu reduzir a quantidade de branches **(-30%)** executadas pelo programa mesmo que um outro loop tenha sido inserido. Entretanto, devido ao fato da iteração percorrer o vetor repetidas vezes, prejudicamos o fator de localidade que poderia ter sido melhor aproveitado e isso prejudicou drasticamente na quantidade de referências à cache **(+58%)**, além de demonstrar um aumento no tempo de execução em comparação ao caso base **(+13%)**.



```
float sum_vectorized(float* v, int size){
    int i;
    int jump = 4;
    __m128 v1, v2;
    while(jump != size){
        for(i = 0; i < size; i += 2*jump){
            v1 = _mm_loadu_ps(v+i);
            v2 = _mm_loadu_ps(v+i+jump);
            v1 = _mm_add_ps(v1, v2);
            _mm_store_ps(v+i, v1);
        }
        jump = 2*jump;
    }
    v1 = _mm_loadu_ps(v);
    v1 = _mm_hadd_ps(v1, v1);
    v1 = _mm_hadd_ps(v1, v1);
    _mm_store_ss(v, v1);

    return v[0];
}

float sum_blocks(float* v){
    // Divide o vetor V em blocos de tal
    // forma que o mesmo caiba completamente
    // na memória cache L1.
    float soma = 0;
    for(int block = 0; block < N; block += BLOCK_SIZE){
        soma += sum_vectorized(v+block, BLOCK_SIZE);
    }
    return soma;
}
```

**Imagem 4** - otimização do código vetorizado utilizando técnicas de blocagem para garantir uma melhor performance da cache devido ao princípio de localidade.

No entanto, isso também nos trouxe à mente uma melhoria que poderia ser executada: a de dividir o vetor em blocos do tamanho da Cache L1 de Dados e passar esses blocos na nossa função vetorizada. Os resultados deste experimento também foram considerados como o caso **SSE3 + Blocagem** nas tabelas anteriores e o código está retratado na imagem desta sessão (*imagem 4*) .

Após aplicar as melhorias, verificamos que conseguimos manter a otimização na quantidade de branches obtida **(-32%)** pela experiência com funções intrínsecas além de garantir um acesso linear ao vetor que não aumentasse drasticamente a quantidade de referências à cache **(-1,7%)**.

Por fim, reparamos que a variação nos parâmetros estudados no caso em que o GCC era utilizado estava muito baixa e, após maior investigação, percebemos que o código fonte gerado tanto pelo caso Base quanto pelo caso otimizado eram iguais, e portanto isso deixou claro que, por mais que as otimizações automáticas sejam bastante úteis, elas nem sempre são capazes de generalizar\* todos os casos em que um código pode ser otimizado.

\* Aparentemente, o website do GNU possui uma lista que enumera os tipos de loops considerados 'vectorizable' pelo compilador e percebemos que o nosso caso não se enquadrava (<https://gcc.gnu.org/projects/tree-ssa/vectorization.html>) em nenhum dos formatos definidos no site. No entanto achamos um outro código muito próximo do nosso mas não tivemos tempo hábil para refazer os experimentos e comparar os possíveis resultados.