# Análise da execução do algoritmo CFRK

**Fabrício Vilasbôas**[1], **Carla Osthoff**[1], **Kary Ocaña**[1],
**Oswaldo Trelles**[2], **Ana Tereza Vasconcelos**[1]

[1]Laboratório Nacional de Computação Científica

[2]Universidad de Málaga

***Resumo.*** *Este trabalho apresenta uma análise do desenvolvimento do **CFRK**, um algoritmo determinístico para uma aplicação de bioinformática computacionalmente intensiva, o k-mer, para uma arquitetura GPGPU. Nossos experimentos demonstram que o **CFRK** é uma alternativa eficiente e de baixo custo para a contabilização de k-mers para análises em metagenoma.*

## 1. Introdução

A grande massa de dados gerada pelas novas tecnologias de sequenciamento [Shendure and Ji 2008] trouxeram um forte impulso nas pesquisas de metagenoma. Por outro lado, o aumento da massa de dados faz necessário o desenvolvimento de técnicas de alto desempenho de baixo custo para a interpretação dos mesmos.

Uma tarefa básica e amplamente usada para análise de amostras de metagenoma [Wooley et al. 2010] é o cálculo da frequência de repetição de *k-mers*. O termo *k-mer* é utilizado como referência a todas as possíveis combinações de comprimento $k$ que estão contidas em uma sequência de dados arbitrária e por sua natureza combinatória demanda muito poder de processamento. Esse tipo de problema se enquadra perfeitamente nas características da arquitetura das *GPU's*. São múltiplos dados operados por uma instrução por vez, caracterizando uma arquitetura *SIMD (Single Instruction Multiple Data)*. A contribuição deste trabalho é a análise do desenvolvimento de um algoritmo de contabilização de *k-mer* determinístico, o **CFRK** (**C**ontagem da **F**requência de **R**epetição de ***k**-mers*) [Vilasboas et al. 2016], para uma arquitetura *GPU*.

## 2. Trabalhos Relacionados

Existem diversos algoritmos para contabilização da frequência de repetição de *k-mers* desenvolvidos para ambientes *multicore* e *manycore*. Entre os algoritmos de contabilização de *k-mer* desenvolvidos para ambientes *manycore*, *GPU*, o algorítimo **Gerbil** é o de melhor desempenho e mais próximo ao desempenho **CFRK**.

O **Gerbil** [Erbert et al. 2017] é um algoritmo para contabilização da frequência de repetição de *k-mers* com suporte a múltiplas *GPU's* e que faz uso do disco para melhorar a eficiência em relação à memória. Sua execução consiste de duas fases: distribuição e contabilização. Este algoritmo é especializado no processamento de genomas. Isto significa que ele considera que toda subsequência contida no arquivo de entrada pertence a um mesmo organismo e ao final do processamento obtém-se apenas um vetor de contabilização para todo o arquivo de entrada. O **CFRK** foi desenvolvido para o processamento de metagenoma. Isto significa que ele considera que cada subsequência pertence a um organismo distinto e ao final do processamento obtém-se um vetor de contabilização

para cada subsequência do arquivo de entrada. Por este motivo o **Gerbil** é capaz de processar valores de $k < 32$ e o **CFRK** processa valores de $k \leq 8$, que são os valores mais utilizados para análise de metagenoma. Além de analisar amostrar de metagenoma o **CFRK** é também capaz de analisar amostras de genoma enquanto que o **Gerbil** é somente capaz de analisar amostras de genoma.

## 3. Descrição do Algoritmo CFRK

Em genômica computacional, *k-mer* é o conjunto de todas as possíveis subsequências de comprimento $k$ a partir de uma leitura obtida através do sequenciamento do material genético. Dada uma sequência de comprimento $L$, o valor dos possíveis *k-mers* é definido pela Equação 1a, enquanto o número de possíveis combinações de *k-mers* de um conjunto de dados, quando $n$ corresponde ao número de valores possíveis para cada dado ($4$ no caso do material genético) é definido pela Equação 1b. O cálculo do número de *k-mers* é uma etapa importante nos processos de montagem e de alinhamento de sequências.

$$nComb_{k-mer} = L - k + 1 \tag{1a}$$

$$nComb_{total} = n^k \tag{1b}$$

Os arquivos de entrada para o **CFRK** são oriundos do sequenciamento de metagenomas. Estes arquivos possuem inúmeros fragmentos do material genético, chamados de *reads*, e cada *read* é composto por uma sequência de nucleotídeos.

Os *reads* ao serem lidos do disco são agrupados na memória em caixas denominadas *chunks*. Esses *chunks* são enfileirados e enviados um a um para o processamento na *GPU*. Com isto pode-se ajustar de forma dinâmica a quantidade de dados sendo enviada para a *GPU* e, assim, tem-se o controle da quantidade de memória utilizada para o processamento. Os *chunks* possuem um tamanho fixo definido pelo usuário.

## 4. Resultados e análise

Foi utilizado para os teste uma máquina com um processador Intel(R) Xeon(R) CPU X5650 @ 2.67GHz com 12 núcleos, memória RAM de 50 GB, sistema operacional CentOS 7, versão do kernel 3.10.0-514.2.2.el7.x86_64, uma *GPU Nvidia Tesla K40* com 2888 núcleos e 12 GB de memória RAM.

Foi utilizada uma amostra real de metagenoma sequenciada por um *Illumina HiSeq2000*. Essa amostra possui a identificação *SRX2021688* no banco de dados *SRA* do *NCBI* (*National Center for Biotechnology Information*). Este é um tipico arquivo de grande massa de dados para análise de metagenoma contendo 8.7GB de dados.

### 4.1. Resultado do perfilador

Para a obtenção destes resultados foi utilizado o nvprof [NVidia 2017]. O nvprof é um perfilador via linha de comando disponibilizado com o próprio *toolkit* do *CUDA* [Kirk and Wen-mei 2012].

As Tabelas 1, 2, 3 e 4 apresentam a saída do perfilador. A primeira coluna apresenta o nome dos *kernels*; a segunda coluna apresenta a ocupação que é a relação entre a média de *warps* ativos por ciclo e a quantidade máxima de *warps* suportada pela placa; a

terceira coluna apresenta o IPC (*Instructions per cycle*) que é a quantidade de operações executadas por ciclo; a quarta coluna apresenta a Eficiência por SM (*Streaming Multiprocessor*) que é o percentual de tempo que pelo menos um *warp* ficou ativo em algum SM; a quinta coluna apresenta as Instruções por *warp* que é a média de instruções executadas por *warp*; a sexta coluna apresenta *Warps* por ciclo que é a quantidade média de *warps* ativos por ciclo.

| Kernel | Ocupação | IPC | Eficiência por SM | Inst. por Warp | Warps por ciclo |
|---|---|---|---|---|---|
| SetMatrix | 72.7992% | 2.624794 | 66.066352% | 27.000000 | 7.612246 |
| ComputeIndex | 92.1302% | 4.082246 | 98.011027% | 202.124624 | 15.704332 |
| ComputeFreq | 54.1978% | 0.955904 | 98.901368% | 24.687500 | 2.402939 |

**Table 1: Dados do perfilador para $k = 2$**

| Kernel | Ocupação | IPC | Eficiência por SM | Inst. por Warp | Warps por ciclo |
|---|---|---|---|---|---|
| SetMatrix | 70.9859% | 2.755066 | 81.414651% | 27.000000 | 7.403433 |
| ComputeIndex | 93.7637% | 4.003148 | 98.465103% | 281.627854 | 17.681540 |
| ComputeFreq | 53.5372% | 0.948170 | 98.917671% | 24.687500 | 2.374436 |

**Table 2: Dados do perfilador para $k = 3$**

| Kernel | Ocupação | IPC | Eficiência por SM | Inst. por Warp | Warps por ciclo |
|---|---|---|---|---|---|
| SetMatrix | 69.6877% | 2.878994 | 90.703457% | 27.000000 | 7.588800 |
| ComputeIndex | 94.8903% | 3.950892 | 98.828498% | 361.110782 | 17.960260 |
| ComputeFreq | 51.3717% | 0.863294 | 99.068069% | 24.687500 | 2.150786 |

**Table 3: Dados do perfilador para $k = 4$**

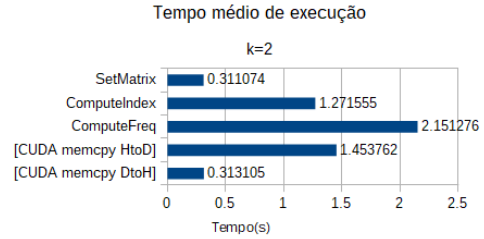
| Kernel | Ocupação | IPC | Eficiência por SM | Inst. por Warp | Warps por ciclo |
|---|---|---|---|---|---|
| SetMatrix | 69.2726% | 2.893869 | 94.382579% | 27.000000 | 7.559753 |
| ComputeIndex | 94.5527% | 4.031779 | 98.886495% | 440.578637 | 17.933021 |
| ComputeFreq | 48.0829% | 0.707771 | 99.188304% | 24.687500 | 1.815714 |

**Table 4: Dados do perfilador para $k = 5$**

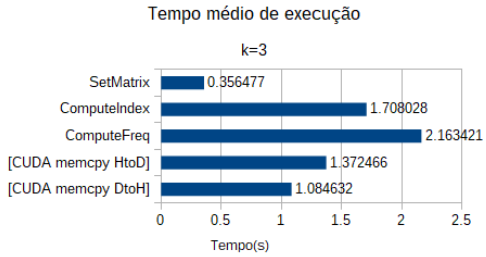
As Figuras 1a, 1b, 1c e 1d foram geradas a partir do comando *nvprof –csv -t 600*. Esse comando retorna duas classes de resultados. A primeira classe refere-se as informações do *runtime* do *CUDA*. A segunda classe refere-se a *API* do *CUDA*. Os resultados mostrados nas Figuras citadas acima neste parágrafo são referentes apenas ao *runtime* do *CUDA*. O esquema de cores das Figuras 1a, 1b, 1c e 1d apresenta o percentual de execução dos *kernels* de forma que segue um padrão onde: azul representa a mais demandante, laranja representa a segunda mais demandante, amarelo representa a terceira mais demandante, verde representa a quarta mais demandante e vermelha representa a menos demandante.

### 4.1.1. Discussão dos resultados

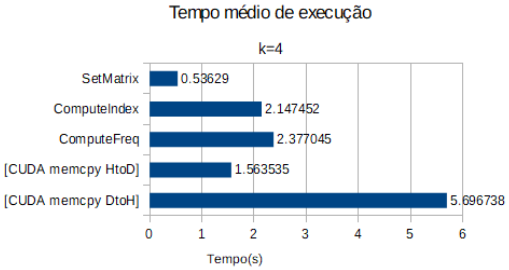
Podemos observar que o tempo das rotinas ”Leitura/Mapeamento” e ”Cópia Host-Device” dependem apenas do tamanho do arquivo de entrada. As rotinas ”SetMatrix”



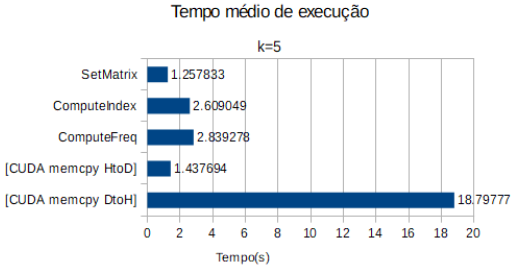

**(a)** Tempo médio de execução para $k = 2$




**(b)** Tempo médio de execução para $k = 3$




**(c)** Tempo médio de execução para $k = 4$




**(d)** Tempo médio de execução para $k = 5$

**Figure 1: Tempo médio de execução das funções do *runtime* da GPU**

e ”ComputeIndex” aumentam de complexidade com o valor de k, como mostrado pela Equação 2, onde $nS$ é o número de sequências e $4^k$ é o número total de combinações. O aumento do valor de $k$, aumenta o tamanho do vetor de frequência da rotina ”SetMatrix” e consequentemente o número de operações de conversão na rotina ”ComputeIndex”. A rotina ”ComputeFreq” diminui a sua complexidade com o aumento de $k$, devido à diminuição do numero de combinações possíveis, conforme a Equação 1a. Desta forma, a medida em que se aumenta o valor de $k$, a rotina de ”Cópia Device-Host” torne-se o *hotspot*, seguida pelas rotinas ”SetMatrix”, ”ComputeIndex” e ”ComputeFreq”, e, por fim, a etapa ”Cópia Host-Device”, conforme as Figuras 1a, 1b, 1c e 1d.

$$\alpha = nS * 4^k \tag{2}$$

Podemos observar que apesar da complexidade da execução da rotina ”ComputeFreq” diminuir com o aumento do valor de $k$ o tempo de processamento não diminui. Isto se deve ao fato desta rotina estar executando operações de escrita em memória em uma região crítica, gerando a execução das operações de escrita de forma serial, demandando um número maior de ciclos de processamento. Este fato pode ser comprovado ao observar os valores da coluna IPC das Tabelas 1, 2, 3 e 4. Enquanto o valor de IPC da rotina ”ComputeIndex” mantem-se acima de 4, o valor da rotina ”ComputeFreq” tende a zero, ou seja, enquanto a rotina ”ComputeIndex” realiza 4 operações por ciclo a rotina ”ComputeFreq” realiza menos de uma operação por ciclo.

Ao analisarmos as Tabelas 1, 2, 3 e 4 observamos que a rotina ”SetMatrix” apresenta pouca variação em Ocupação, IPC e em *Warps* por ciclo, mantendo uma média de $69\%$, $2.8$ e $7.5$, respectivamente. O valor de Inst. por *Warp* é constante e igual a $27$. O valor de Eficiência por SM varia de $66.066352\%$ a $94.382579\%$, mostrando uma

melhora na utilização dos *SM's*. A rotina "ComputeIndex" apresenta a melhor escalabilidade, possui uma Ocupação mínima de $92.1302\%$ com $k = 2$ e uma Ocupação máxima de $94.8903\%$ com $k = 4$. Possui o valor mínimo de Eficiência por SM de $98.011027\%$ com $k = 2$ e valor máximo de $98.886495\%$ com $k = 5$. Os valores de IPC e de *Warp* por ciclo não possuem muita variação, mantendo uma media de $4$ instruções por ciclo e $18$ *Warps* por ciclo, respectivamente. Observamos também um aumento significante do número de instruções por *Warp* a medida em que $k$ aumenta, de $281.627854$, para $k = 2$, a $440.578637$, para $k = 5$. Observamos também que a rotina "ComputeFreq" demanda muitos ciclos conforme os valores da coluna IPC das Tabelas 1, 2, 3 e 4. Enquanto o valor de IPC da etapa "ComputeIndex" se mantém acima de $4$, o valor da etapa "ComputeFreq" tende a zero, isto significa que a rotina "ComputeIndex" realiza $4$ operações, enquanto a rotina "ComputeFreq" realiza menos de uma. Isto é causado pela execução de operações em região crítica.

## 5. Conclusão e trabalhos futuros

Neste trabalho apresentamos a análise da execução do **CFRK** e observamos uma boa eficiência para os *kernels* implementados: para a etapa "SetMatrix" foi observada uma eficiência mínima de $69.27\%$ para $k = 5$ e máxima de $72.79\%$ para $k = 2$. Para a etapa "ComputeIndex" foi observada uma eficiência mínima de $92.13\%$ para $k = 2$ e máxima de $94.89\%$ para $k = 4$. Para a etapa "ComputeFreq" foi observada uma eficiência mínima de $48.08\%$ para $k = 5$ e máxima de $54.19\%$ para $k = 2$.

Como trabalho futuro pretendemos avaliar o ganho de desempenho através da utilização da memória de textura e da memória compartilhada. Observamos também que a medida em que aumentamos o valor de k, a rotina de transferência de dados entre a memória principal e a placa *GPU* cresce de tal forma que domina o tempo de processamento para $k = 5$. Como trabalhos futuros pretendemos avaliar o ganho no tempo de processamento com a utilização da tecnologia *NVLINK* que deverá diminuir o gargalo de transferência de dados entre as memórias da *CPU* e da *GPU*. Por final, pretendemos estender a implementação do **CFRK** para ambientes de múltiplas *GPU's* de forma a possibilitar o processamento de arquivos de entrada maiores que o tamanho da memória da placa *GPU*.

## References


Erbert, M., Rechner, S., and Müller-Hannemann, M. (2017). Gerbil: a fast and memory-efficient k-mer counter with gpu-support. *Algorithms for Molecular Biology*, 12(1):9.

Kirk, D. B. and Wen-mei, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.

NVidia (2017). Profiler user's guide.

Shendure, J. and Ji, H. (2008). Next-generation dna sequencing. *Nature biotechnology*, 26(10):1135–1145.

Vilasboas, F., Osthoff, C., Trelles, O., and Vasconcelos, A. T. (2016). Otimização de um algoritmo paralelo para contabilização da repetição de k-mers. *II Escola Regional de Computação de Alto Desempenho do Rio de Janeiro*.

Wooley, J. C., Godzik, A., and Friedberg, I. (2010). A primer on metagenomics. *PLoS Comput Biol*, 6(2):e1000667.