Ferramenta para análise de sequenciamento mRNA de célula única (scRNA-seq)

Arthur pereira da Fonseca

1

1. Introdução

Células são consideradas as unidades básicas da vida, os tijolos que formam o organismo. O corpo humano, por exemplo, é formado por cerca de 37 trilhões de células. Compostas de vários tipos celulares de diferentes formas, tamanhos e funções. E tudo isso é determinado pela diferença na expressão gênica de cada célula.

Para estudar essa expressão genética, utiliza-se sequenciamento do RNA mensageiro (mRNA), usado para a fabricação de proteínas [Han et al. 2015]. Nos métodos tradicionais de sequenciamento de expressão de mRNA, utiliza-se um pedaço de tecido, ou uma população celular, e sequencia-se tudo como um inteiro. Assim, você estará mitigando essa variedade de expressão entre células, e acaba com uma média de expressão de toda aquela amostra [Han et al. 2015].

Em contraste, com o sequenciamento de mRNA de célula única, ou single cell RNA sequencing (scRNA-seq), você consegue preservar a informação de expressão individual. Isso porque as células são processadas individualmente, e não em uma mistura. E, ao final da análise, você obtém perfis de expressão para cada uma dessas células distintas. O que revela a heterogeneidade no tecido ou população celular amostrada. E também mostra a variabilidade na expressão genômica em todas as células da amostra [Saliba et al. 2014].

A metodologia de sequenciamento de mRNA de célula única tem se desenvolvido rapidamente nos últimos anos. Avanços significativos foram feitos tanto no número de células sequenciadas em cada estudo, como também na diminuição dos preços desse tipo de experimento. Para se ter uma ideia, o número de células sequenciadas subiu, em cerca de dez anos, de apenas dez células por estudo realizado, completamente manual, para centenas de milhares de células por estudo.

Ao final de cada estudo é gerado então uma matriz esparsa contendo os dados brutos, que inclui todas as células analisadas e o nível de expressão para cada gene. Essa matriz pode ser interpretada contendo as células como entidades e sua expressão gênica como seus atributos. E, conforme aumentamos o número de células por estudos, aumentamos também o tamanho dessas matrizes [Stegle et al. 2015].

Para entender e obter informações significativas desses dados, se faz necessário uma longa etapa de análise, que inclui filtragem celular, normalização, redução de dimensão, clusterização e detecção de marcadores moleculares [Stegle et al. 2015].

Com a metodologia tradicional, utilizando Python, leva-se várias horas para concluir todas essas análises. O objetivo desse trabalho, portanto, é criar uma ferramenta capaz de clusterizar os dados normalizados em um tempo hábil. Para isso foram empregadas técnicas de paralelização em GPU, utilizando CUDA, e em CPU, com OpenMP.

2. Algoritmo

Antes de se iniciar a clusterização das células é necessário que os dados estejam normalizados. A forma mais simples de fazer essa normalização é com o algoritmo de contas por milhão (CPM). Nele, somamos o valor total de expressão para cada célula, e em seguida normalizamos cada dado de expressão gênica da célula dividindo o valor dessa expressão pelo total de expressão da célula, e multiplicamos o resultado por um milhão. No entanto, esse algoritmo assume que todas as células possuem níveis semelhantes de expressão gênica, o que não é verdade.

Para clusterizar as células, na tentativa de agrupar os mesmos tipos celulares considerando sua expressão gênica, é necessário ou aplicar uma técnica de redução de dimensionalidade, ou empregar um *framework* capaz de trabalhar com essa matriz larga e esparsa gerada pelo sequenciamento de célula única. Nesse sentido, o algoritmo escolhido foi o Sparse MinMaxKMeans, proposto por Sayak Dey [Dey et al. 2020]. Esse algoritmo combina o algoritmo MinMaxKMeans de Tzortzis e Likas[Tzortzis and Likas 2014] com o *framework* seleção de atributos em matrizes esparsas por Witten e Robert [Witten and Tibshirani 2010].

A primeira etapa do algoritmo consiste na função SparseKMeans representada no pseudo código 1. A função recebe a matriz contendo as células a serem clusterizadas e o número de clusters desejados.

SparseKMeans:
Input: data matrix n*p and number of clusters k
Output: Clusters centers and objects memberships.

1: Initialize column weights to $1/\sqrt{p}$

2: Initialize clusters with **MinMaxKMeans**

3: **WHILE** (column wheights convergence is greater than 1e-4) do:
4: **UpdateCenters** (column weights, matrix)

5: **UpdateWeights** (clusters, matrix)

6: Colum wheights convergence = $\frac{colum\ weights - old\ column\ weights}{old\ column\ weights}$

Figure 1. Pseudo código da função SparseKMeans.

A função inicia executando o algoritmo MinMaxKMeans com a matriz inteira. Em seguida, entra-se em um *loop* no qual serão atualizados o centro dos clusters de acordo com o peso de cada coluna da matriz, na função UpdateCenters; e o peso de cada coluna na função UpdateWeights. A função termina quando os pessoas associados às colunas convergem para zero.

O algoritmo MinMaxKMeans está representado pelo pseudo código 2. Ele começa distribuindo os centros de forma aleatória, caso não sejam previamente fornecidos. O que diferencia esse algoritmo do KMeans tradicional é que ele visa minimizar a máxima variância intra clusters. Para isso, cada cluster recebe um peso que está associado com o número de membros que ali estão. Dessa forma, clusters muito cheios ou vazios

possuem um grande impacto sobre o algoritmo. Em seguida calcula-se a distância euclidiana de cada ponto para cada centro dos cluster, porém considerando o peso associado aos clusters para realizar o cálculo dessas distancias. Define-se a população de cada cluster, e move os centros para a média das distâncias dos objetos de cada cluster. Em seguida os pesos são atualizados e o loop é repetido até que não haja grandes mudanças de clusters ou até um número máximo de loops.

```
MinMaxKMeans
Input: data matrix n*p and number of clusters k or clusters centers
Output: Clusters centers and objects memberships.

1: IF (inputed number of clusters): initialize random centers

2: Initialize cluster weights to 1 / numclusters

3: WHILE (new cluster weights – old clusterwheigts > 1e-04) do:
4:      Weight distances according to cluster weights
5:      Find nearest center for each object
6:      Move centers to cluster means
7:      Calculate euclidean distances from each object to each center
8:      Update cluster weights
```

Figure 2. Pseudo código da função MinMaxKMeans.

Para atualizar os centros, com a função UpdateCenters, utiliza-se um subset da matriz considerando apenas as colunas com peso maiores que zero. Para gerar essa submatriz, os valores de cada coluna são modificados de acordo com o peso associado. Em seguida move-se os centros considerando esses novos valores, e repetimos a etapa de MinMaxKMeans para essa submatriz, dessa vez utilizando os centros já definidos.

Na atualização dos pesos com a função UpdateWeights, utiliza-se a matriz inteira para gerar uma escala dos objetos de cada cluster e uma escala de todos os objetos da matriz. Essa escala é uma lista de tamanho igual ao número de colunas da matriz. Calcula-se então a diferença entre essas duas escalas e o valor de *lam* dessa diferença. Os novos pesos são então definidos como o valor de cada item, dessa lista de diferença entre as escalas, dividido por *lam*.

```
UpdateCenters
Input: matrix n*p and column weights
Output: Clusters centers and objects memberships.

1: Get subset of matrix were column weights != 0

2: Move centers considering non zero wheights

3: MinMaxKMeans(subset matrix, centers)
```

```
UpdateWeights
Input: matrix n*p and clusters
Output: columns weights

1: wCSS = Get cluster scales

2: tCSS =Get full matrix scale

3: diffArray = Diference betwen two scales

4: lam = calculate lam for diffArray

5: Result = Difference / lam
```

Figure 3. Pseudo código das funções UpdateWeights e UpdateCenters.

3. Abordagem

Para fazer a paralelização da normalização utilizando CPM foi empregado OpenMP. Tendo em vista que é um processo relativamente rápido, levando menos de um segundo em alguns casos, o custo de se copiar a matriz inteira para a placa de vídeo, fazer a normalização e copiar de volta para a CPU seria muito grande. E, por isso, foi escolhida a normalização apenas com OpenMP.

A paralelização do algoritmo de clusterização foi feita de forma híbrida. Como havia disponível apenas uma placa gráfica e a matriz ocupa grande parte, se não toda, a memória da GPU, não foi possível paralelizar o algoritmo inteiro na placa de vídeo. A fim de evitar então o maior número possível de transferências entre a memoria da CPU e da placa, a estratégia utilizada foi paralelizar os cálculos que envolvem a matriz inteira na GPU, sendo eles o MinMaxKMeans executado pela função SparseKMeans e o UpdateWeights em cada loop da mesma.

No entanto, a matriz inteira ainda não coube na placa de vídeo. Por isso foi necessário uma nova estratégia para alcançar a paralelização dessas etapas que utilizam a matriz inteira. E a estratégia escolhida foi mover o máximo possível da matriz para a placa gráfica, executando ali em paralelo. E o restante da matriz sendo processada em paralelo na GPU usando também OpenMP. Ao final da etapa na CPU, o resultado é então juntado ao da GPU, e seguem-se os cálculos na própria GPU.

Foi notado que o cálculo de distancia euclidiana realizado pela função MinMaxKMeans é o que consome mais tempo de execução e, portanto, foi o foco da nossa paralelização. Na GPU a abordagem utilizada foi que cada thread realiza esse cálculo para uma célula contra um cluster, sendo necessário um total de threads igual ao número de células (n) vezes o número de clusters (k). Já na CPU o numero de células foi dividida entre as threads na hora de realizar esse cálculo.

Para o cálculo dos centros na função UpdateCenters, que não pôde ser paralelizado com GPU devido à memória, foi utilizado uma paralelização também com OpenMP. Note que para isso, é preciso primeiro copiar os pesos calculados na GPU para a CPU a fim de se chegar na submatriz. Em seguida resolve-se o MinMaxKMeans para essa submatriz também em paralelo, mas agora com o OpenMP.

4. Dificuldades

Uma grande dificuldade encontrada na paralelização desse trabalho foi realizar a redução em paralelo, tanto com CUDA quanto com OpenMP. Apesar do OpenMP oferecer uma diretiva para a redução de um único valor, isso não ocorre no caso das listas. Em ambos os casos foram feitas tentativas para solucionar esse problema mas sem sucesso. Ao final, a solução encontrada foi ou rodar a etapa em sequencial, ou, no caso da GPU, baixar esses dados para a CPU para fazer a redução utilizando OpenMP. Isso com certeza tem um grande impacto sobre o *speedup* e o nível de desempenho obtidos.

Outro problema encontrado foi a utilização das *tasks* em OepnmMP. Como não obtive sucesso em implementar as *tasks*, optei por iniciar um conjunto de threads sempre que necessário, o que implica em *overheads* muito grandes. Isso é claramente visível na função UpdateCenters, conforme será mostrado na seção a seguir, em que o aumento de numero de threads faz com que o *overhead* seja tão grande que o tempo gasto se torna

maior do que na execução sequencial.

Por fim, algo que também não ficou muito bom, e que pode ser melhorado, é explorar os blocos e toda a capacidade da GPU na hora de definir os *grids*. Da forma como foi feito, foram escolhidos os *grids* mais simples para execução.

5. Desempenho

Os testes de desempenho a seguir foram executadas em uma máquina Linux 64-bit, de 16 CPUs, com processador AMD Ryzen 7 e equipada com uma placa de vídeo GeForce RTX 2060 com 5,9GB de memória. Para os testes foram utilizados dados de expressão de células T do sangue e líquido sinovial disponíveis no Expression Atlas [EBI] contendo 77.035 células por 21.123 genes.

5.1. Normalização

Como dito anteriormente, a normalização foi paralelizada apenas com OpenMP tendo em vista que normalizar toda a matriz leva menos de um segundo utilizando 16 threads conforme mostrado na tabela 1. Note, no entanto que a paralelização utilizando CPM não é a mais recomendada. De fato, ao utilizar ela notamos que o algoritmo leva mais tempo para ser executado, pois leva mais tempo para achar os clusters. Sendo assim, todos os testes a seguir foram realizados sem a normalização, utilizando os dados já normalizados disponíveis no expression atlas.

Table 1. Tempo gasto, speedup e eficiência para a normalização de diferentes tamanhos de matriz com diferentes números de threads.

Threads Células	1	2	4	8	16	32	
15.000	1,36	0,64	0,38	0,27	0,17	0,15	Tempo
	1,00	2,13	3,58	5,04	8,00	9,07	Speedup
	1,00	1,06	0,89	0,63	0,50	0,28	Eficiência
30.000	2,17	1,34	0,80	0,44	0,30	0,29	Tempo
	1,00	1,62	2,71	4,93	7,23	7,48	Speedup
	1,00	0,81	0,68	0,62	0,45	0,23	Eficiência
60.000	5,35	2,34	1,24	0,79	0,57	0,58	Tempo
	1,00	2,29	4,31	6,77	9,39	9,22	Speedup
	1,00	1,14	1,08	0,85	0,59	0,29	Eficiência
77.035	6,94	3,54	1,82	1,15	0,72	0,70	Tempo
	1,00	1,96	3,81	6,03	9,64	9,91	Speedup
	1,00	0,98	0,95	0,75	0,60	0,31	Eficiência

5.2. Sparse MinMaxKMeans

Os testes a seguir foram realizados utilizando diferentes tamanhos de matriz e número de clusters igual a 7. Para fins de comparação, também foi elaborado um código paralelizando todo o algoritmo apenas em OpenMP.

Conforme podemos ver na tabela 2 nós obtivemos grandes desempenhos utilizando GPU, chegando a alcançar speedups até maiores do que 100x. Porém, cerca de 65 mil células apenas couberam na memória da GPU, criando um obstáculo para a paralelização, já que os outros 12 mil restantes tiveram que ser paralelizados na CPU, confomre discutido na seção *Abordagem*. Nota-se uma queda muito grade do speedup

de 100 para 16 vezes quando isso ocorre. Não deixando, porém, de ser um ganho significativo, e um tempo bem melhor quando comparado com a versão sequencial ou com a paralelização apenas em CPU. Essa paralelização com OpenMP apenas, manteve um speedup constante ao variar apenas o número de células.

Table 2. Tempo total de execução para diferentes tamanhos de matriz na versão OpenMP e na versão híbrida. Ambas utilizando 7 clusters e 8 threads.

Células	Tempo Sequencial	OpenMP	Sepedup	CUDA+OpenMP	Speedup
15.000	1.497,99	431,00	3,48	15,04	99,60
30.000	4.085,45	1.181,00	3,46	31,58	129,37
60.000	8.256,37	2.473,00	3,34	61,86	133,47
77.035	1.699,99	472,49	3,60	104,24	16,31

Quando quebramos o algoritmo em partes para analisar separadamente as diferentes etapas, podemos ver os grandes ganhos proporcionados pela placa de vídeo no cálculo do MinMaxKMeans com a matriz inteira e no calculo dos pesos. Na tabela 3 foram medidos o desempenho para cada etapa do algoritmo em uma versão paralelizada apenas com OpenMP, e a versão com paralelização mista utilizando CUDA e OpenMP.

Table 3. Tempo de execução gasto em cada etapa do algoritmo para a versão OpenMP e versão híbrida. Todos os testes foram feitos na matriz completa e com 7 clusters.

Processadores:		2		4		8		16	
Etapa	Tempo Sequencial	OpenMP	Sepedup	OpenMP	Sepedup	OpenMP	Sepedup	OpenMP	Sepedup
MinMaxKMeans	1280,85	666,28	1,92	340,53	3,76	309,00	4,15	279,00	4,59
UpdateCenters	11,42	13,07	0,87	14,29	0,80	25,00	0,46	61,00	0,19
UpdateWeights	378,02	202,26	1,87	111,77	3,38	120,00	3,15	171,00	2,21
Tempo Total	**1.699,99**	**900,40**	**1,89**	**485,37**	**3,50**	**472,49**	**3,60**	**530,00**	**3,21**
	Tempo Sequencial	CUDA+OpenMP	Sepedup	CUDA+OpenMP	Sepedup	CUDA+OpenMP	Sepedup	CUDA+OpenMP	Sepedup
MinMaxKMeans	1280,85	122,50	10,46	76,86	16,66	52,80	24,26	50,09	25,57
UpdateCenters	11,42	22,32	0,51	20,38	0,56	20,29	0,56	69,71	0,16
UpdateWeights	378,02	33,38	11,32	19,86	19,03	11,75	32,17	24,94	15,16
Tempo Total	**1.699,99**	**197,12**	**8,62**	**136,48**	**12,46**	**104,24**	**16,31**	**164,20**	**10,35**

Notamos que ao aumentar o número de processadores, tanto na paralelização mista como na com apenas OpenMP obtemos ganhos de speedup apenas até cerca de 10 processadores. Isso por conta do *overhead* de criar esse número de threads. Conforme discutido na seção *Dificuldades*, não foi possível fazer uma implementação com *tasks*, sendo necessário passar por esse overhead a cada chamada da função. Este *overhead* fica bem claro no cálculo de atualização dos centros, e isso está associado aos vários *loops* chamados pela funçãoMinMaxKMeans dentro dessa função. Sendo que em cada loop você deve passar por todo o processo de gestão de threads, limitando assim os nossos ganhos ao aumentar o número de processadores.

6. Resultado

Para fim de comparação, os clusters gerados foram comparados com o disponível no Expression Atlas. A figura 4 ilustra essa diferença. Os dados foram gerados utilizando 7

clusters e número máximo de 20 loops. É possível perceber certa semelhança, porém o algoritmo empregado é mais exigente quanto ao preenchimento de todos os clusters, fazendo com que outros clusters apareçam. Seria necessário no entanto uma análise de marcadores moleculares para entender as diferenças geradas e comparar a um nível molecular as duas abordagens.

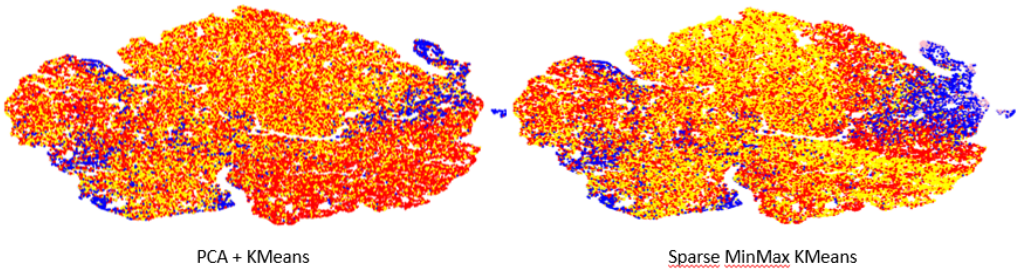

Figure 4. Resultados obtidos pela clusterização empregada nesse trabalho em comparação com os dados do expression atlas.

7. Conclusão

No fim foi possível obter um ótimo desempenho, com speedup de até 100x quando a matriz cabe inteira na memória da GPU, comparado com apenas 3x com a paralelização utilizando apenas OpenMP.

Quando os dados não cabem na memoria da placa de vídeo, e é preciso recorrer a uma paralelização mista para as funções mais lentas, há uma perda significativa nos nossos ganhos de speedup. Porém ainda há um ganho de até 16x quando combinado com o processamento em OpenMP com 8 cores.

No entanto, os vários problemas discutidos na seção *Dificuldades*, implicam em uma grande perda de poder de processamento e aumento do *overhead*. Isso é facilmente notado pelo aumento de tempo gasto e diminuição do speedup observado na tabela 3. Mais evidentemente no tempo gasto com a função UpdateCenters, que chega a demorar mais tempo do que com a versão sequencial, devido ao alto custo associado a criação das threads em cada loop interno da função.

Os possíveis problemas já estão elencados, e agora podem ser estudados a fim de se encontrar uma solução e obter uma melhor performance na paralelização.

8. Compilação

Para facilitar, os três programas gerados são compilados a partir de um único arquivo. Dentro da pasta tp_final_Arthur, utilize os seguinte comando para compilar as versões sequencial, OpenMP apenas e OpenMP mais CUDA, respectivamente:

```
gcc -std=c99 -o sequential Sequential/sequential_one_file.c -lm

gcc -std=c99 -fopenmp -o openmp OpenMP/openmp_one_file.c -lm

nvcc  -Xcompiler " -fopenmp" -o cuda CUDA/cuda_one_file.cu
```

Para executar todas as versões é necessário indicar o arquivo mtx contendo a matriz a ser analisada. Uma matriz de 1.000 células acompanha esse relatório.

Para a versão sequencial deve-se dizer se deseja ou não executar a normalização em CPM passando "yes" ou "no" como segundo argumento. Dessa forma, para rodar a versão sequencial sem normalização basta utilizar o comando:

```
./sequential sample/sample_1000cells_normalised.mtx no
```

Para ambas as versões OpenMP e CUDA deve-se também especificar o número de threads desejadas. Para isso basta indicar o número de threads no segundo argumento, e se deseja a normalização no terceiro. Por exemplo:

```
./openmp sample/sample_1000cells_normalised.mtx 8 no
```

Executa a versão OpenMP com 8 threads sem normalização. E:

```
./cuda sample/sample_1000cells_normalised.mtx 16 yes
```

Executa a versão CUDA com 16 threads e com normalização.

References

Dey, S., Das, S., and Mallipeddi, R. (2020). The sparse minmax k-means algorithm for high-dimensional clustering. In Bessiere, C., editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 2103–2110. International Joint Conferences on Artificial Intelligence Organization. Main track.

EBI, G. E. T. Single cell expression atlas.

Han, Y., Gao, S., Muegge, K., Zhang, W., and Zhou, B. (2015). Advanced applications of RNA sequencing and challenges. *Bioinformatics and Biology Insights*, 9s1:BBI.S28991.

Saliba, A.-E., Westermann, A. J., Gorski, S. A., and Vogel, J. (2014). Single-cell RNA-seq: advances and future challenges. *Nucleic Acids Research*, 42(14):8845–8860.

Stegle, O., Teichmann, S. A., and Marioni, J. C. (2015). Computational and analytical challenges in single-cell transcriptomics. *Nature Reviews Genetics*, 16(3):133–145.

Tzortzis, G. and Likas, A. (2014). The minmax k-means clustering algorithm. *Pattern Recognition*, 47(7):2505–2516.

Witten, D. M. and Tibshirani, R. (2010). A framework for feature selection in clustering. *Journal of the American Statistical Association*, 105(490):713–726.