Universidade Estadual de Campinas

MO644

Introdução à Programação Paralela

Paralelização do Algoritmo de Esqueletização Zhang-Suen Thinning com CUDA e Clang

---

Autor:	*RA:*
Darwin Ttito Concha	192618

June 26, 2017

Contents

1 Descrição do Algoritmo **2**

2 Descrição da Região Sequential que foi Paralelizada **3**

3 Paralelização da Região Sequential **4**

4 Análise de Speedup **5**

5 Dificuldades **7**

1 Descrição do Algoritmo

Este algoritmo de esqueletização é um método paralelo que significa que o novo valor obtido depende apenas do valor anterior da iteração. Este algoritmo é feito por duas sub-iterações. No primeiro, um pixel $P(i, j)$ é excluído se as seguintes condições forem satisfeitas:

1. Seu número de conectividade é um.
2. Tem pelo menos dois vizinhos pretos e não mais de seis.
3. Pelo menos um de $P(i, j + 1), P(i - 1, j)$ e $P(i, j - 1)$ são brancos.
4. Pelo menos um de $P(i - 1, j), P(i + 1, j)$ e $P(i, j - 1)$ são brancos.

Na segunda sub-iteração, as condições nos passos 3 e 4 mudam.

1. Seu número de conectividade é um.
2. Tem pelo menos dois vizinhos pretos e não mais de seis.
3. Pelo menos um de $P(i - 1, j), P(i, j + 1)$ e $P(i + 1, j)$ são brancos.
4. Pelo menos um de $P(i, j + 1), P(i + 1, j)$ e $P(i, j - 1)$ são brancos.

No final, os pixels que satisfaçam estas condições serão excluídos. Se, no final de qualquer sub-iteração, não haja pixels a serem excluídos, então o algoritmo pára.

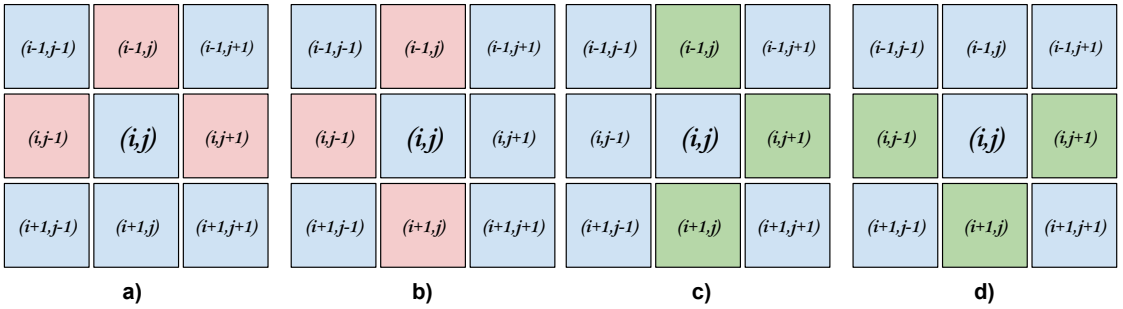


Figure 1: Nas imagem a) e b), os pixels de cor vermelho correspondem a os pixels das condições 3 e 4 da primeira sub-iteração. Nas imagem c) e d), os pixels de cor verde correspondem a os pixels das condições 3 e 4 da segunda sub-iteração

2 Descrição da Região Sequential que foi Paralelizada

Na *Figura 2* mostra a sequência de etapas que o algoritmo têm que seguir, descrevendo com palavras de cor azul os métodos da implementação que pertencem para cada etapa.

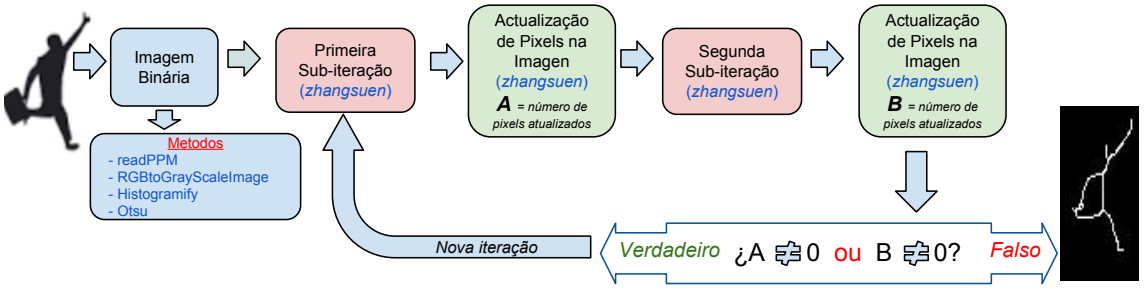


Figure 2: Sequência de etapas do algoritmo.

time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.97	**40.07**	**40.07**	**1**	**40.07**	**40.07**	**zhangsuen**
0.07	40.10	0.03	1	0.03	0.03	RGBtoGrayScaleImage
0.07	40.13	0.03	1	0.01	0.01	main
0.02	40.14	0.01	1	0.00	0.00	Histogramify
0.00	40.14	0.00	1	0.00	0.00	rtclock
0.00	40.14	0.00	1	0.00	0.00	Otsu
0.00	40.14	0.00	1	0.00	0.00	readPPM
0.00	40.14	0.00	1	0.00	0.00	writePPM

Table 1: Profiling, Gprof.

De acordo com o profiling obtido podemos observar que o método zhangsuen consome a maior parte do tempo da execução total (etapas vermelhas e verdes concordo com a Figura 2), portanto, esta será o método para ser paralelizado.

As etapas de cor vermelho(primeira e segunda sub-iteração) verifica se para cada pixel é satisfeita as condições 1,2,3,4 apresentadas na descrição do algoritmo, enquanto as etapas de cor verde faz a atualização dos pixels concordo com as etapas de cor vermelho. Embora todas as etapas pertencem ao mesmo método(zhangsuen), cada etapa depende do resultado da etapa anterior, portanto, foi decidido fazer a paralelização individualmente para cada etapa.

3 Paralelização da Região Sequential

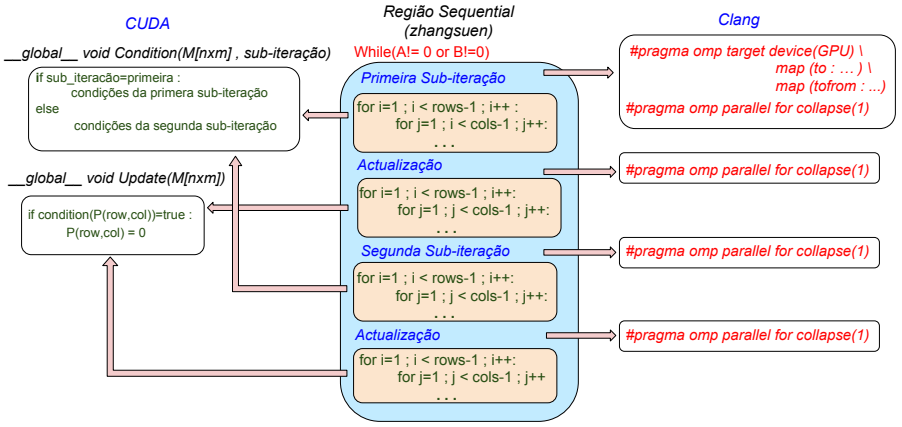


Figure 3: região sequencial paralelizada com CUDA e Clang.

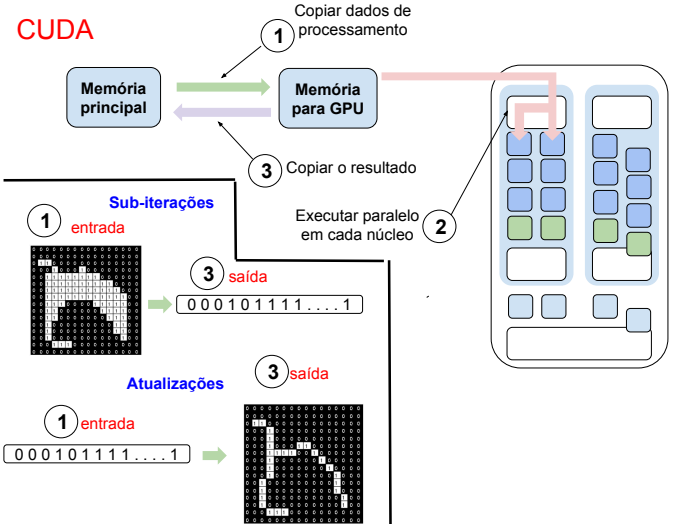


Figure 4: Paralelização com CUDA.

Como se mostra na Figura 3, a paralelização é realizada em as sub-iteração e atualizações antes mencionados. Cada thread da GPU é responsável por uma dada posição da matriz de entrada, então em CUDA as sub-iterações são substituídos pelo método __global__ Condition, enquanto as atualizações são substituídos pelo metodo __global__ Update. Em Clang é utilizado o construtor "target" para transferir o controle do host para o device, bem como para estabelecer um ambiente de dados do device, para depois paralelizar as sub-iterações e atualizações como mostra a Figura 3.

4 Análise de Speedup

Foram utilizados 9 imagens de tamanhos diferentes(em formato .ppm) para fazer os testes e análise do speedup.

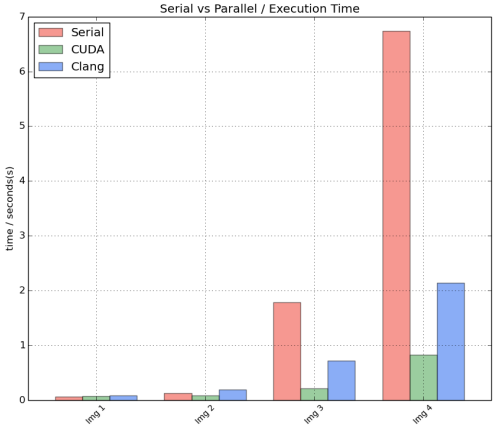
Input (imagem)	**Tamanho(pixels)**
1	225x225
2	400x224
3	700x490
4	1500x1192
5	1920x1920
6	3175x2068
7	4200x4200
8	5000x6675
9	10328x7760

Table 2: Imagens de teste.

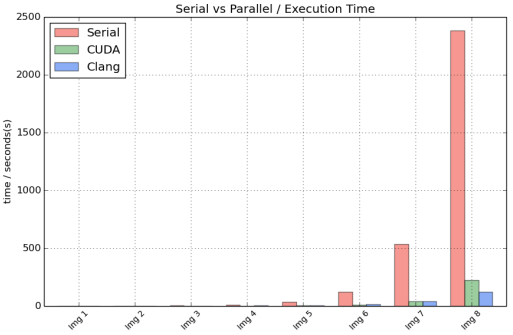
A paralelização foi feita usando CUDA e Clang. Os tempos de execução e speedup para os dois métodos estão mostrados na Tabela 3. A comparação entre o tempo de execução paralelo(CUDA e Clang) e serial são mostrados através de gráficos de barras na Figura 5. A Comparação entre os tempos de execução paralelo de Clang e CUDA são mostrados na Figura 6.

Input	Tempo de Execução							Speedup	
	Serial	CUDA					Clang	CUDA	Clang
		Create buffer	offload send	kernel	offload receive	total			
1	0.082	0.053	0.004	0.001	0.003	*0.072*	0.081	1.139	1,012
2	0.121	0.048	0.014	0.015	0.007	*0.085*	0.114	1.424	1,061
3	1.784	0.044	0.072	0.047	0.045	*0.209*	0.714	8.535	2,498
4	6.736	0.038	0.434	0.120	0.237	*0.830*	2.136	8.116	3,154
5	36.220	0.054	1.735	0.322	0.891	*3.004*	4.812	12.057	6,696
6	121.908	0.037	5.418	1.128	2.804	*9.388*	15.176	12.986	8,033
7	532.340	0.048	22.663	4.034	11.605	*38.350*	41.476	13.881	12,835
8	2383.162	0.029	136.443	17.926	70.303	*224.703*	120.355	10,606	19,801
9	9877.854	0.0406	444.256	72.817	225.575	*742.688*	654.077	13.300	15.101

Table 3: Tempo de execução serial e paralelo, cálculo do speedup.

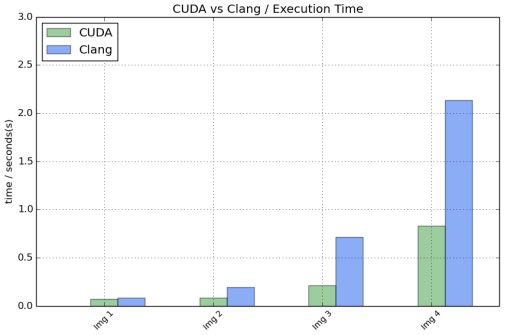


(a) Primeiras 4 imagens.

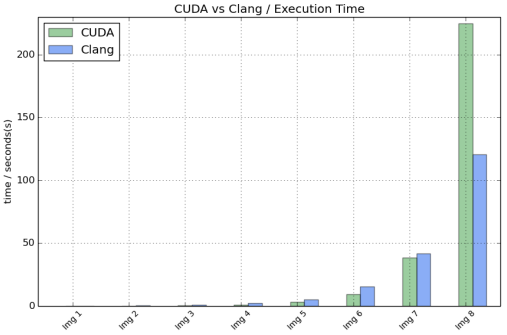


(b) Todas as imagens.

Figure 5: Tempo de execução serial vs paralelo.



(a) Primeiras 4 imagens.



(b) Todas as imagens.

Figure 6: CUDA vs Clang.

De acordo com a Tabela 3 e Figura 5, claramente Clang e CUDA demonstram um speedup considerável em imagens de grandes dimensões ($\geq$ 1500x1192 pixels).

De acordo com a Tabela 3 e Figura 6 podemos observar que CUDA é claramente melhor que Clang nas primeiras 7 imagens de teste, obtendo como melhor speedup 13.881 na imagem 7 . No entanto, Clang prova ser muito melhor que CUDA nas últimas duas imagens de teste (imagens de alta resolução) obtendo o melhor speedup de todas os testes na imagem 8 (speedup=19,801 , ver Tabela 3).

5 Dificuldades

Uma dificuldades surgiu ao tentar modularizar o método __global__ de CUDA, por isso, decidimos não usar modularidade.
Outra dificuldade foi quando tentamos paralelizar os métodos de sub-iterações e atualizações em um só método __global__, dificuldade que não foi resolvido, e por isso optamos por paralelizar ambos métodos por separado, por causa de sua dependência de resultados.