Quantificação de Análise de Recorrência Eficiente na GPU

Lucas Froguel
Programação Paralela em GPUs

1 Introdução

Nesse trabalho foi feita uma implementação em paralelo na GPU do algoritmo descrito no artigo enviado ao professor. A ideia básica do algoritmo lá descrito é calcular a laminariedade usando pequenos blocos da matriz total chamados de microestados. Dentro de cada microestado, montamos um diagrama das linhas horizontais e calculamos a laminaridade por:

$$\mathrm{LAM} = \frac{\sum_{v=v_{\min}}^{K} vP(v)}{\sum_{v=1}^{K} vP(v)} \tag{1}$$

Assim, a ideia da implementação em GPU é fazer com que cada bloco processe um microestado, de modo que possamos ter o processamento simultâneo de vários microestados. O funcionamento do código está explicado nos comentários, mas aqui segue um resumo do código do kernel.

1. Definir e alocar espaços

```
__shared__ float mapx[Q], mapy[Q];
__shared__ bool microstate[Q * Q];
__shared__ int histogram[Q];
int tid = threadIdx.x;
int gtid = blockDim.x * blockIdx.x + threadIdx.x;
int i, j;

// get two random indexes
i = xrand[blockIdx.x];
j = yrand[blockIdx.x];
```

2. Carregar os dados a serem processados da memória global para a shared memory

```
// load data [i, i+Q] and [j, j+Q] to shared memory
// each thread loads [tid, tid+blockDim, ...] until all elements are loaded
for (int k = tid; k < Q; k = k + blockDim.x){
   // i, j are such that i + Q, j + Q < N
   mapx[k] = d_map[i + k];
   mapy[k] = d_map[j + k];
   // clean histogram on shared memory
   histogram[k] = 0;
}
```

3. Calcular o microestado

```
// calculate CRP / microstate
for (int k = tid; k < Q * Q; k = k + blockDim.x){
   bool m = 0;
   int iq = k % Q, jq = k / Q;
   float val = abs(mapx[iq] - mapy[jq]);
   if (val < e) m = 1;
   microstate[k] = m;
}
```

4. Montar o histograma

```
// each thread will look for lines in one row of the microstate
// values will be CRP[tid, k], k\in[0, Q-1]
// assumes Q < threadsPerBlock (which is very reasonable, given the amount
    of shared memory per block)
if (tid < Q){
   int line_length = 0;
   for (int k = 0; k < Q; k++){
      if (microstate[k + tid * Q] == 1){
         line_length += 1;
         if (k == Q - 1){
            atomicAdd(&histogram[line_length-1], 1);
            line_length = 0;
         }
      }
      else if (line_length > 0 && microstate[k + tid * Q] == 0){
         atomicAdd(&histogram[line_length-1], 1);
         line_length = 0;
      }
   }
}
```

5. Fazer a soma da Eq.(1)

```
// should be a cheap/fast operation, but has to be done on one thread
if (tid == 0){
   float LAM = 0, total = 0;
   for (int k = 0; k < Q; k++){
      if (k + 1 >= lmin){
         LAM += histogram[k] * (k+1);
      }
      total += histogram[k] * (k+1);
   }
   if (total != 0) d_LAM[blockIdx.x] = LAM / total;
}
```

Para a CPU tudo que resta é gerar números aleatórios, carregar e transferir dados para a GPU

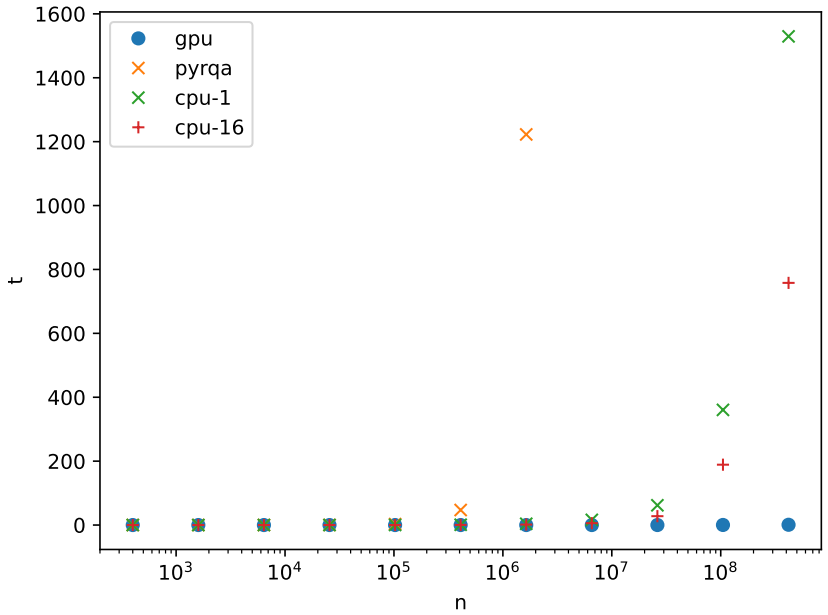

Figura 1: Figura com os tempos de cálculo das 4 implementações consideradas

e iniciar o kernel.

2 Resultados

Para avaliar a eficiência, os resultados da GPU foram comparados com duas implementações no mesmo algoritmo na cpu (uma usando apenas 1 thread e outra usando a versão paralela), ambas em julia, e também com o PyRQA, que é uma implementação paralela na GPU do algoritmo tradicional (e usa OpenCL).

A máquina utilizada possuí um i7-10700KF (com 16 threads) e uma RTX3060. As informações detalhadas de ambas estão no repositório na pasta PC-data.

Os resultados estão mostrados na Fig.(1) e na Tab.(1). O maior speedup, curiosamente, foi com $ind = 6$ contra o PyRQA e foi de $\sim 3,2 \times 10^5$ vezes. Contra a cpu, foi em $ind = 10$ e foi de cerca de ~ 2000 vezes. Isso demonstra que a implementação em GPU da nova abordagem é extremamente mais rápida que a implementação em GPU existente e do que as de CPU. Apesar de ambos os resultados serem esperados, pois esse algoritmo é, de fato, mais rápido e a GPU é realmente mais rápida do que a CPU.

ind	n	time_gpu	time_cpu	time_cpu_multi	time_pyrqa	gpu_min_speedup
0	400	0.000044	0.000961	0.049372	0.084893	21.8
1	1600	0.000034	0.003351	0.001733	0.057727	51.0
2	6400	0.000039	0.023179	0.004588	0.085305	117.6
3	25600	0.000086	0.067011	0.024826	0.320447	288.7
4	102400	0.000265	0.231871	0.123956	3.011581	468.0
5	409600	0.000967	0.920062	0.393596	47.047580	407.4
6	1638400	0.003784	3.969902	1.489342	1222.607544	393.9
7	6553600	0.014997	16.227974	5.941865	NaN	396.15
8	26214400	0.060014	61.813792	27.755246	NaN	462.6
9	104857600	0.241047	360.439228	188.912355	NaN	783.9
10	419430400	0.879212	1529.509376	757.977858	NaN	862.3

Tabela 1: Essa tabela mostra os valores de tempo para o calculo da LAM para cada implementação e o speedup mínimo da gpu.