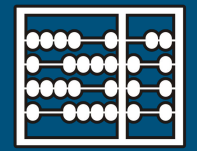
Programação de GPU: Usando a Placa de vídeo para resolver problemas

Lucas de Oliveira Silva aka Marcão - Mestrando em Computação

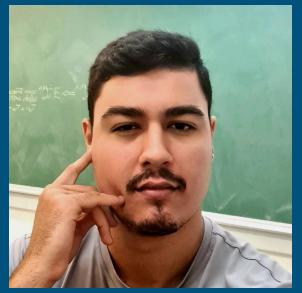
SECOMP 2022

UNICAMP



INSTITUTO DE COMPUTAÇÃO

Sobre mim



- Tecnico em informatica pelo COTUCA (PD15);
- Estagiei por um ano e trabalhei 4 meses como Dev na Dextra (NodeJS);
- Fiz IC em computação distribuída/paralela aplicada a problemas de geofísica;
- No meio da graduação troquei pra uma IC em Machine Learning;
- Formado em Ciência da Computação (CC018);
- Hoje faço mestrado em Algoritmos Parametrizados aplicados ao FTP - problema de Otimização Combinatória;

Por que Programação Paralela?

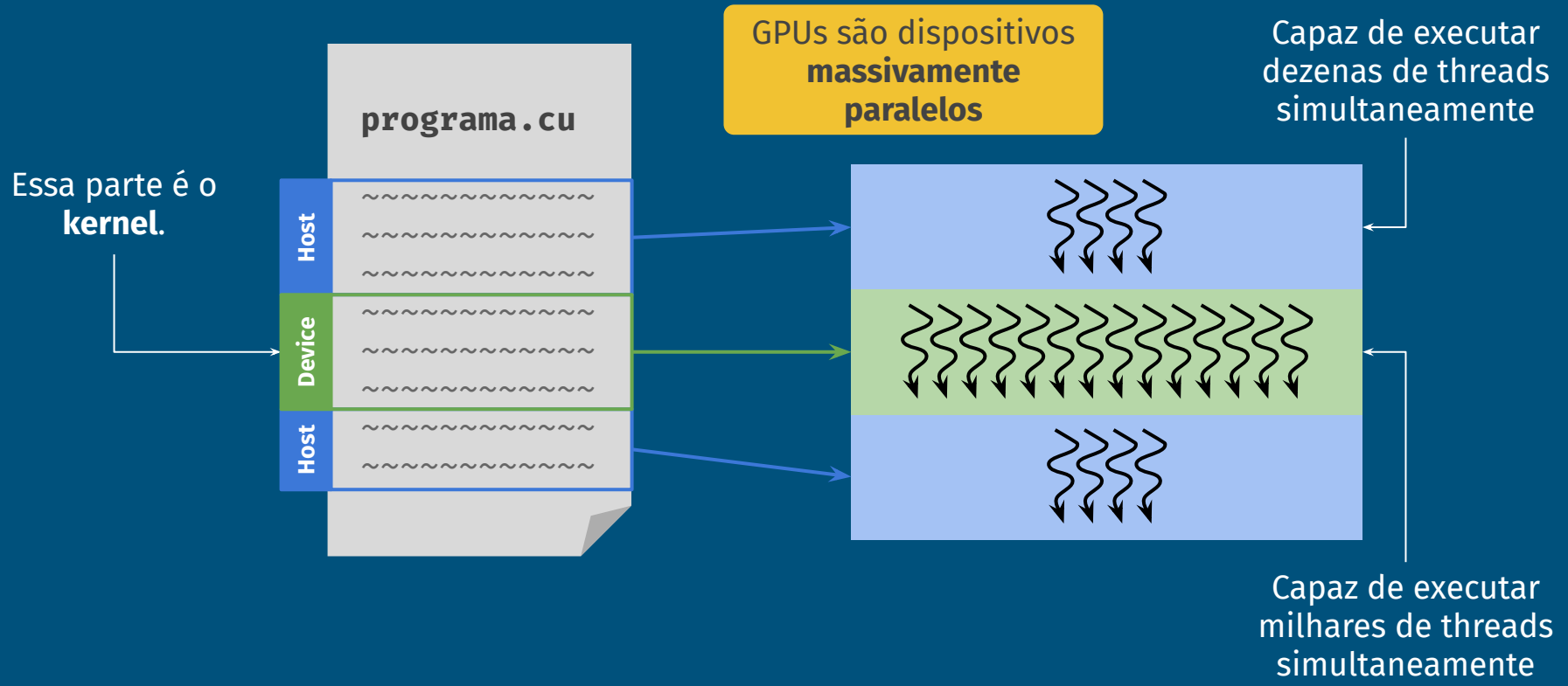
- Hoje há mais poder computacional, mas problemas maiores e mais complexos;
- Desaceleração da taxa de crescimento da performance dos microprocessadores;
- Estamos atingindo os limites físicos para a densidade de transistores;
- Ao invés de uma única CPU com clock altíssimo temos vários cores autônomos;
- Não adianta nada adicionar cores se os programadores não sabem tirar proveito;

O que é CUDA?

Definição

CUDA (Arquitetura de Dispositivo de Computação Unificada) é uma plataforma e API de computação paralela para GPUs Nvidia

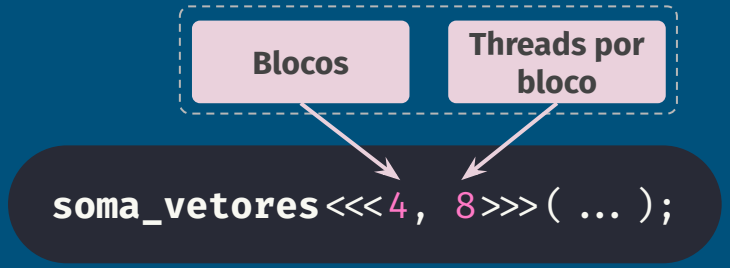
Computação Heterogênea

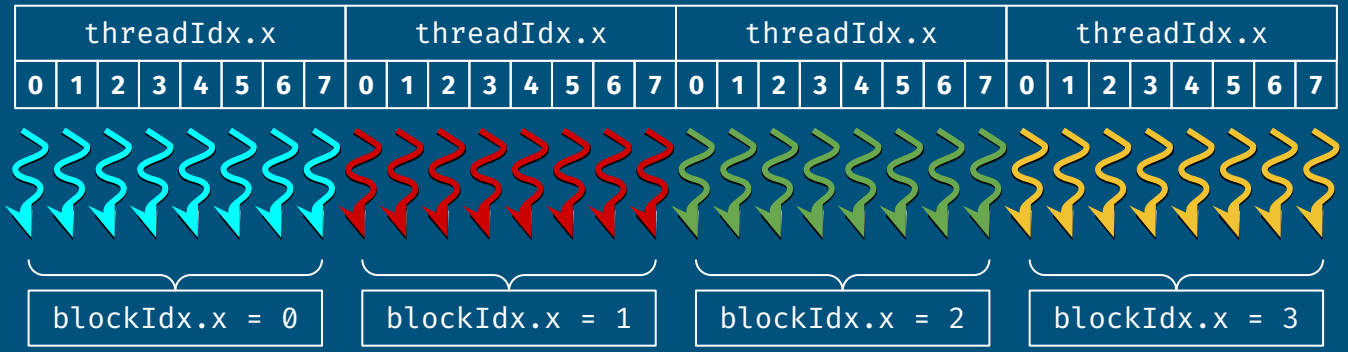


Exemplos Básicos

Notebook Colab

Controle de threads

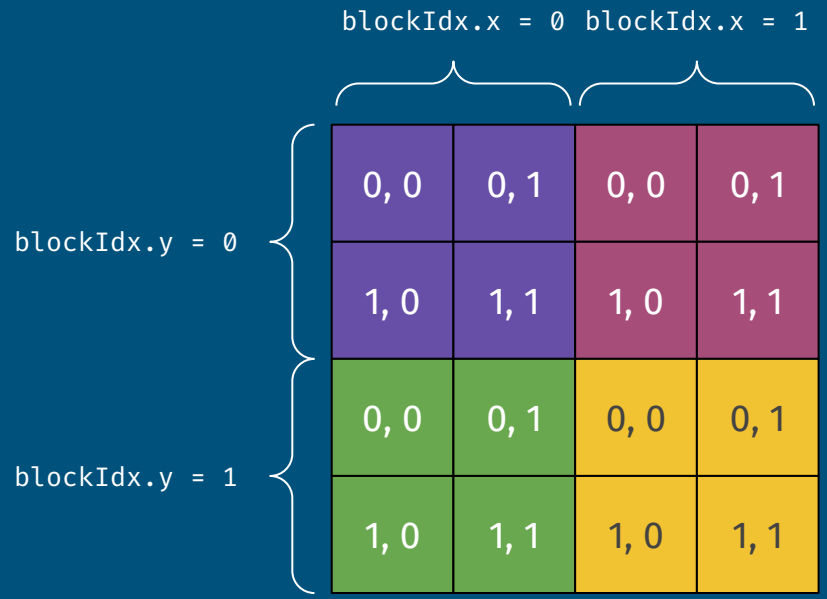




Exemplo de Grid 2D

```
dim3 B = { 2, 2, 1 };
dim3 T = { 2, 2, 1 };

func<<<B, T>>>( ... );
```



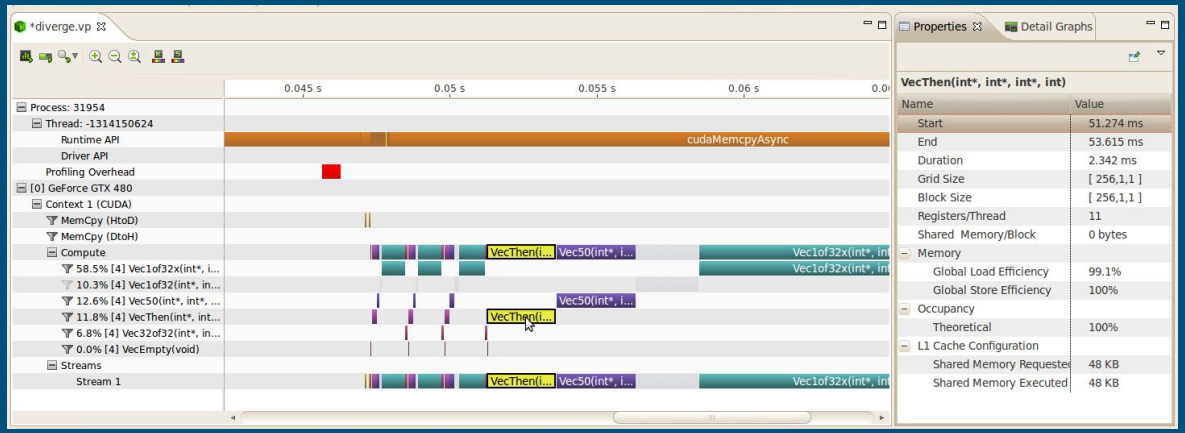
Análise de desempenho

- Speedup = (Tempo da versão base)/(Tempo da versão alterada);
- No melhor dos casos temos speedup linear no número de cores;
- Usar ferramentas de **profiling** para encontrar os **hot spots**;
- As mais simples são **perf** para profiling na CPU e **nvprof/Nsight** para GPU;

```
Samples: 3K of event 'cycles:u', Event count (approx.): 1884268461
Overhead  Command         Shared Object         Symbol
  51.84%  soma_vetores_0  soma_vetores_0        [.] soma_vetores
  38.19%  soma_vetores_0  soma_vetores_0        [.] main
   9.88%  soma_vetores_0  [unknown]             [k] 0xffffffff9be0109c
   0.02%  soma_vetores_0  ld-linux-x86-64.so.2  [.] 0x000000000000c4d2
   0.02%  soma_vetores_0  ld-linux-x86-64.so.2  [.] 0x00000000000121e0
   0.02%  soma_vetores_0  ld-linux-x86-64.so.2  [.] 0x000000000000be43
   0.02%  soma_vetores_0  ld-linux-x86-64.so.2  [.] 0x000000000000bb85
   0.01%  soma_vetores_0  ld-linux-x86-64.so.2  [.] 0x000000000001b05a
```

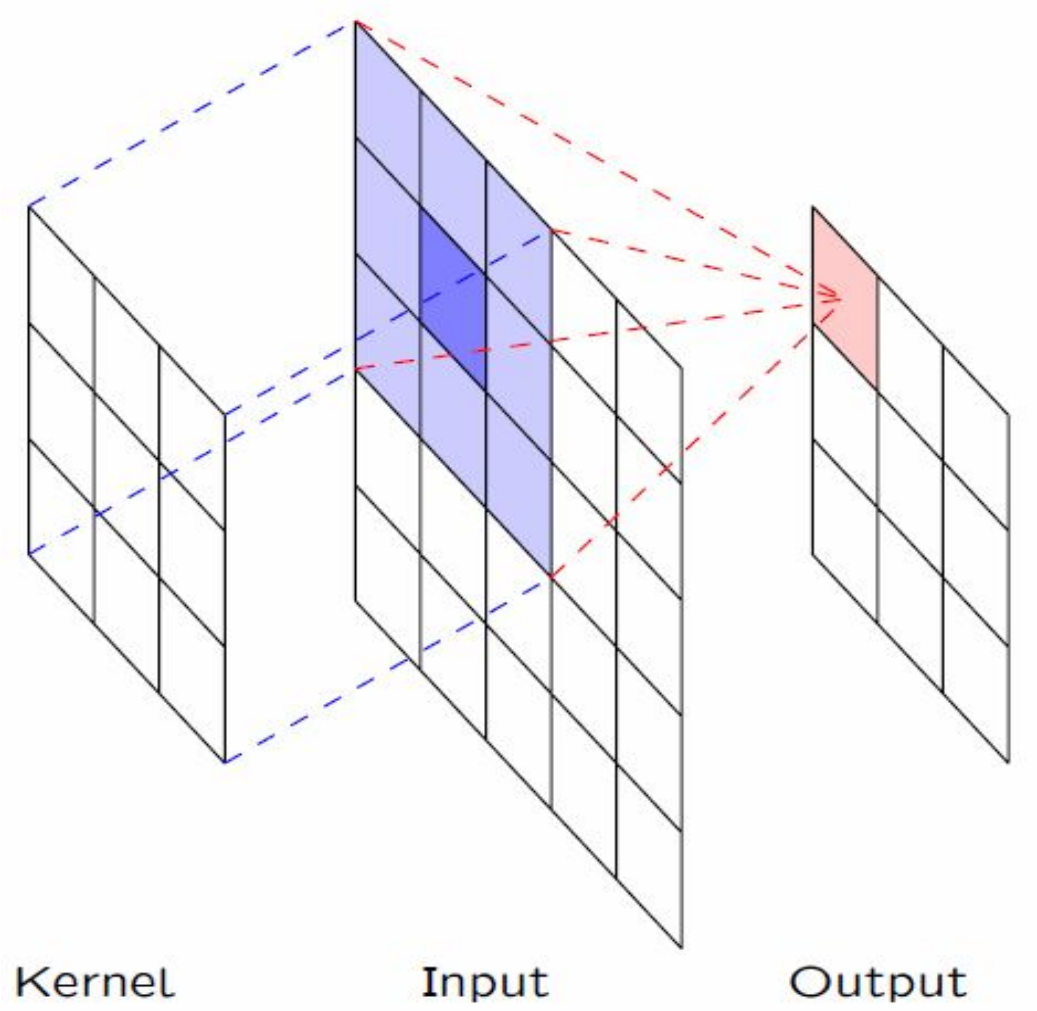
```
Percent |          mov     %rsi,-0x20(%rbp)
        |          mov     %rdx,-0x28(%rbp)
        |          mov     %ecx,-0x2c(%rbp)
        |          movl    $0x0,-0x4(%rbp)
        |        ↓ jmp     70
   0.74 |    20:   mov     -0x4(%rbp),%eax
   0.17 |          cltq
   0.11 |          lea     0x0(,%rax,4),%rdx
  10.07 |          mov     -0x20(%rbp),%rax
   0.63 |          add     %rdx,%rax
  13.23 |          movss   (%rax),%xmm1
   0.23 |          mov     -0x4(%rbp),%eax
   5.35 |          cltq
   0.80 |          lea     0x0(,%rax,4),%rdx
   4.19 |          mov     -0x28(%rbp),%rax
        |          add     %rdx,%rax
   7.98 |          movss   (%rax),%xmm0
   0.68 |          mov     -0x4(%rbp),%eax
   3.01 |          cltq
   0.11 |          lea     0x0(,%rax,4),%rdx
   6.13 |          mov     -0x18(%rbp),%rax
   0.73 |          add     %rdx,%rax
  23.36 |          addss   %xmm1,%xmm0
  11.65 |          movss   %xmm0,(%rax)
   1.07 |          addl    $0x1,-0x4(%rbp)
   1.13 |    70:   mov     -0x4(%rbp),%eax
        |          cmp     -0x2c(%rbp),%eax
   8.64 |        ↑ jl      20
```

```
==8489== NVPROF is profiling process 8489, command: ./soma_vetores_2
out[0] = 3.000
A soma funcionou!
Tempo gasto: 29.00ms
==8489== Profiling application: ./soma_vetores_2
==8489== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   56.20%  246.15ms         1  246.15ms  246.15ms  246.15ms  [CUDA memcpy DtoH]
                   40.96%  179.40ms         2  89.702ms  89.183ms  90.222ms  [CUDA memcpy HtoD]
                    2.83%  12.409ms         1  12.409ms  12.409ms  12.409ms  soma_vetores(float*, float*, float*, int)
      API calls:   53.43%  536.48ms         3  178.83ms  493.38us  535.46ms  cudaMalloc
                   42.52%  426.88ms         3  142.29ms  89.392ms  247.04ms  cudaMemcpy
                    1.70%  17.077ms         1  17.077ms  17.077ms  17.077ms  cudaLaunchKernel
                    1.23%  12.396ms         1  12.396ms  12.396ms  12.396ms  cudaDeviceSynchronize
                    1.04%  10.477ms         3  3.4923ms  585.00us  6.2941ms  cudaFree
                    0.03%  351.29us       101  3.4780us     254ns  213.94us  cuDeviceGetAttribute
                    0.03%  309.19us         1  309.19us  309.19us  309.19us  cuDeviceGetPCIBusId
                    0.01%  61.867us         1  61.867us  61.867us  61.867us  cuDeviceGetName
                    0.00%  3.2070us         3  1.0690us     349ns  2.3750us  cuDeviceGetCount
                    0.00%  1.6100us         2     805ns     295ns  1.3150us  cuDeviceGet
                    0.00%     581ns         1     581ns     581ns     581ns  cuDeviceTotalMem
                    0.00%     572ns         1     572ns     572ns     572ns  cuDeviceGetUuid
                    0.00%     552ns         1     552ns     552ns     552ns  cuModuleGetLoadingMode
```



Exemplo final

Filtro Passa Baixa - Kernel de Convolução 2D


Kernel
Input
Output

Otimização Algorítmica

- Alguns kernels 2D podem ser divididos em dois filtros de uma dimensão;
- Considerando uma imagem M×N e filtro de m×n então a complexidade vai de O(M.N.m.n) para O(M.N.(m+n));
- Se isso é possível o filtro é dito separável, o filtro passa baixa é separável!

$$\frac{1}{3}\begin{bmatrix}1\\1\\1\end{bmatrix} * \frac{1}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix} = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$$

CUDA Streams

Stream é uma sequência de operações executadas na ordem de sua emissão na GPU

CUDA Streams

Síncrono

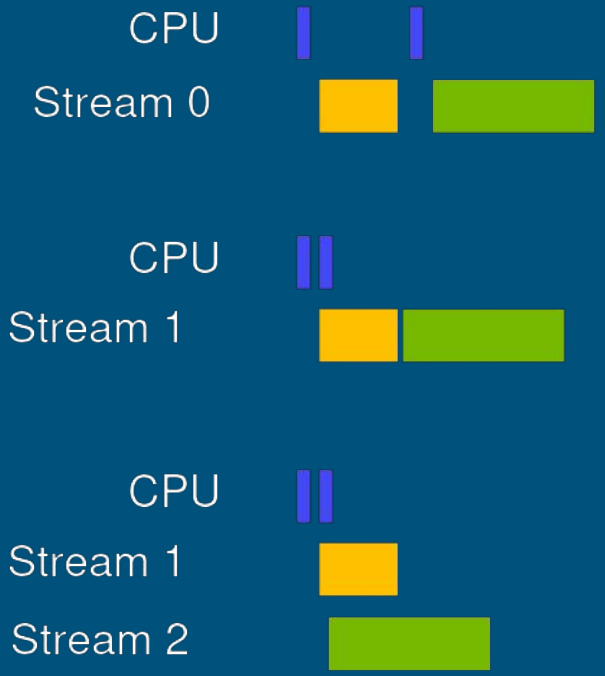
```
cudaMemcpy(...);
foo<<<...>>>();
```

Assíncrono na mesma Stream

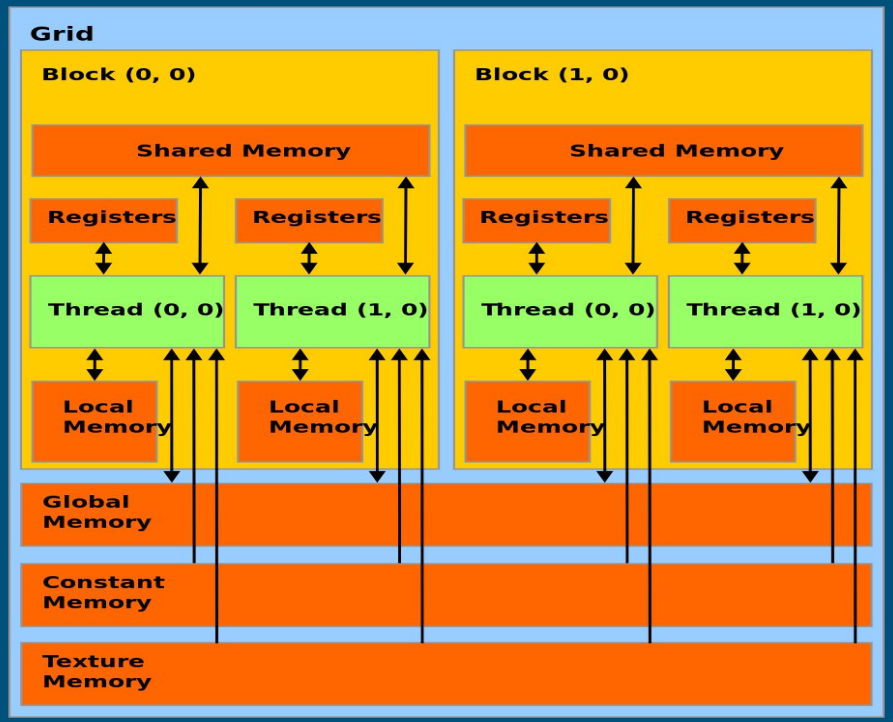
```
cudaMemcpyAsync(..., stream1);
foo<<<..., stream1>>>();
```

Assíncrono em Streams diferentes

```
cudaMemcpyAsync(..., stream1);
foo<<<..., stream2>>>();
```



Shared Memory - o “cache” dos blocos



Dúvidas!?

Referências:

- https://akluz.wordpress.com/mc970-mo644-1s21/ (página de MC970/MO644)
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- https://cuda-tutorial.readthedocs.io/en/latest/
- https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
- https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/
- https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf
- https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf