UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

Lista de Exercícios 2

Disciplina**:** Computação de Alto Desempenho
Prof.: Ricardo Augusto Pereira Franco

***Observações:** Os exercícios são individuais e deverão ser entregues na forma notebook (Colab).

Data de entrega: 23/04/2025.

Exercícios:

1. Descreva como os Processadores de Fluxo (SP) e espaços de memória são organizados numa GPGPU.
2. Como são organizados os *threads*, blocos e grids em CUDA? Mostre como um *kernel* CUDA é invocado em CUDA-C (linguagem C com CUDA).
3. Por que o uso de comandos *if* dentro de *kernels* CUDA pode afetar negativamente o desempenho de aplicações CUDA-C?
4. Implemente um código para inicializar um vetor com 4.000 posições com um valor passado como parâmetro. Eleve todos os valores nos índices pares ao quadrado e os valores nos índices ímpares à terceira potência. Em seguida, implemente uma função sequencial que retorne se as operações foram realizadas com sucesso ou se houve falha. Obs.: número de blocos = 8 e número de *threads* igual a 512 por bloco.
5. Proponha uma solução paralela em CUDA para o cálculo da seguinte expressão vetorial: $\boldsymbol{V}=k1*\boldsymbol{A}+k2*(\boldsymbol{B}+\boldsymbol{C})$, onde $k1$ e $k2$ são constantes definidas pelo programador e $\boldsymbol{A}$, $\boldsymbol{B}$ e $\boldsymbol{C}$ são vetores de tamanho n=5000. Utilize *n_blocos*=8 e *n_threads*=16. Inicialize os vetores $\boldsymbol{A}$ e $\boldsymbol{B}$ de forma paralela e com valores arbitrários (definidos pelo programador). Imprima o vetor $\boldsymbol{V}$.
6. Explique o funcionamento do programa CUDA-C abaixo. Como o desempenho deste programa pode ser melhorado?

```
#include <stdio.h>
const int N = 16;
const int blocksize = 4;
__global__ void kernel (int *data, int *outdata, int N ) {
   int p = 0; int i, ix;
   ix = blockIdx.x * blockDim.x + threadIdx.x;
   for (i = 0; i < N; i++)
      if (data[ix] > data[i])
         p++;
   outdata[p] = data[ix];
}
int main() {
      int a[N] = {15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0};
      int b[N] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
      int isize = N*sizeof(int);
      int i, *ad, *bd;
      cudaMalloc( (void**)&ad, isize );
      cudaMalloc( (void**)&bd, isize );
      cudaMemcpy( ad, a, isize, cudaMemcpyHostToDevice );
      cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );
      kernel<<<N/blocksize, blocksize>>>(ad, bd, N);
      cudaMemcpy( b, bd, isize, cudaMemcpyDeviceToHost );
      cudaFree( ad ); cudaFree( bd );
      for (i = 0; i < N; i++)
         printf("%d ", b[i]);
      printf("\n");
      return EXIT_SUCCESS;
}
```

7. Considere que o programa a seguir deve não apenas iniciar um *kernel* CUDA para adicionar dois vetores em um terceiro vetor, todos alocados com *cudaMallocManaged*, mas também inicializar cada um dos três vetores em paralelo em um *kernel* CUDA.
   Realize experimentos usando *cudaMemPrefetchAsync* no programa abaixo para verificar seu impacto na falha de página e migração de memória.
   a) O que acontece quando você realiza o *prefetch* (pré-busca) em apenas um dos vetores inicializados para o dispositivo (vetor *a*)?
   b) O que acontece quando você realiza o *prefetch* em dois dos vetores inicializados para o dispositivo (vetores *a* e *b*)?
   c) O que acontece quando você realiza o *prefetch* em todos os três vetores inicializados para o dispositivo (vetores *a*, *b* e *c*)?
   d) Crie hipóteses sobre o comportamento da Memória Unificada (UM), falhas de página especificamente, bem como o impacto no tempo de execução relatado do *kernel* de inicialização, antes de cada experimento e, em seguida, verifique executando o comando *nvprof*.
   e) Adicione *prefetch* adicional de volta à CPU para a função que verifica a correção do *kernel addVectorInto*. Novamente, crie uma hipótese sobre o impacto na Memória Unificada (UM) antes de executar o *nvprof* para validar sua hipótese.

Código:

```
#include <stdio.h>
```

```
/*
 * Host function to initialize vector elements. This function
 * simply initializes each element to equal its index in the
 * vector.
 */

void initWith(float num, float *a, int N)
{
  for(int i = 0; i < N; ++i)
  {
    a[i] = num;
  }
}

/*
 * Device kernel stores into `result` the sum of each
 * same-indexed value of `a` and `b`.
 */

__global__
void addVectorsInto(float *result, float *a, float *b, int N)
{
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  int stride = blockDim.x * gridDim.x;

  for(int i = index; i < N; i += stride)
  {
    result[i] = a[i] + b[i];
  }
}

/*
 * Host function to confirm values in `vector`. This function
 * assumes all values are the same `target` value.
 */

void checkElementsAre(float target, float *vector, int N)
{
  for(int i = 0; i < N; i++)
  {
    if(vector[i] != target)
    {
      printf("FAIL: vector[%d] - %0.0f does not equal %0.0f\n", i, vector[i], target);
      exit(1);
    }
  }
  printf("Success! All values calculated correctly.\n");

}

int main()
```

```
{
  const int N = 2<<24;
  size_t size = N * sizeof(float);

  float *a;
  float *b;
  float *c;

  cudaMallocManaged(&a, size);
  cudaMallocManaged(&b, size);
  cudaMallocManaged(&c, size);

  initWith(3, a, N);
  initWith(4, b, N);
  initWith(0, c, N);

  size_t threadsPerBlock;
  size_t numberOfBlocks;

  threadsPerBlock = 1;
  numberOfBlocks = 1;

  cudaError_t addVectorsErr;
  cudaError_t asyncErr;

  addVectorsInto<<<numberOfBlocks, threadsPerBlock>>>(c, a, b, N);

  addVectorsErr = cudaGetLastError();
  if(addVectorsErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString(addVectorsErr));

  asyncErr = cudaDeviceSynchronize();
  if(asyncErr != cudaSuccess) printf("Error: %s\n", cudaGetErrorString(asyncErr));

  checkElementsAre(7, c, N);

  cudaFree(a);
  cudaFree(b);
  cudaFree(c);
}
```