Memória e Sincronização em CUDA

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

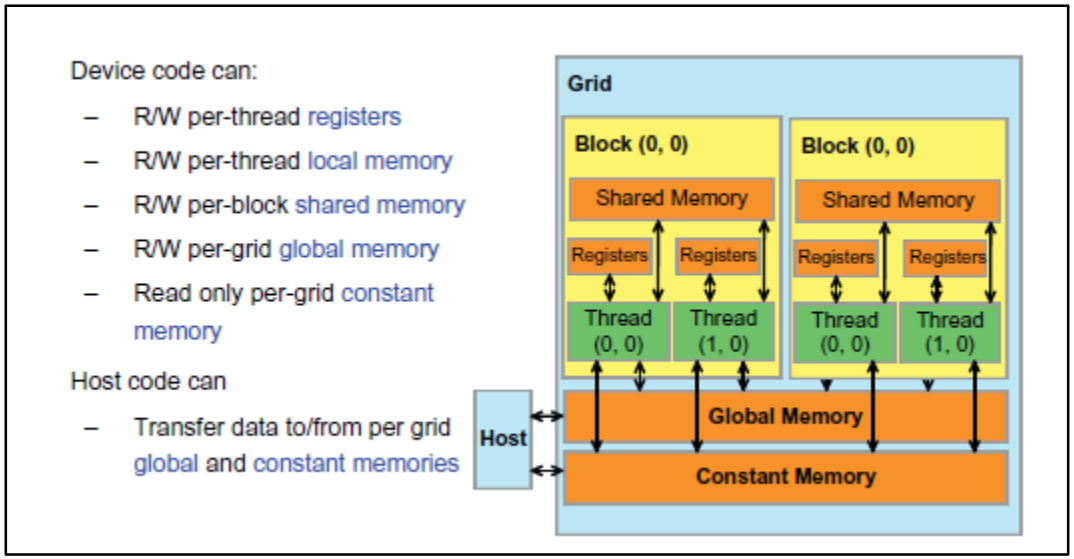
Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP

Memórias em CUDA

- GPUs têm uma sofisticada estrutura de memória para dar suporte à alta vazão de instruções
  - **Memória global por grade/aplicação**
  - **Registradores e memória local à *thread***
  - **Memória compartilhada por bloco**
  - **Memória constante por grade/aplicação**
  - **Memória de textura**

- Transferências com o *host* consideram memórias global e constante



Kirk & Hwu (2016)

Memória Global

- Possui o maior tamanho na GPU mas tem acesso mais lento
- Qualquer *thread* da grade tem acesso à memória global
- Tempo de vida limitado pelo tempo de execução da aplicação
  - Conteúdo das variáveis persistem de uma grade para outra
- São armazenadas na memória global as variáveis que são:
  - Declaradas com o modificador __***device***__
  - Globais no código fonte
    - declaradas fora das funções ou com a diretiva *#define* em C
  - Passadas como argumento das funções de *kernel*
  - Alocadas dinamicamente com o uso do *cudaMalloc( )*

Memória Local

- Variáveis declaradas dentro do *kernel* são locais às *threads* com modificadores *__global__* e *__device__*
- Tempo de vida dessas variáveis limita-se ao tempo de vida do *kernel*
- Preferência ao uso de registradores, a menos que falte espaço ou a variável seja vetor
- Espaço da memória local é menor
- Exemplo: ***tam***, ***i*** e ***j*** estão na memória local,
  matrizA, ***matrizB*** e ***matrizC*** estão na global devido ao ***cudaMalloc( )***

```
#define TAM 100
__global__ void soma(int *matrizA, int *matrizB,int *matrizC, int tam)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j= blockDim.y * blockIdx.y + threadIdx.y;

    if (i < tam && j < tam)
    {
        matrizC[i*TAM+j]=matrizA[i*TAM+j]+matrizB[i*TAM+j];
    }
}
```

Exemplo 01 – ilustra uso da memória local e global

Memória Compartilhada

- *Threads* de um mesmo bloco compartilham esta memória
- Possuem uma latência menor e uma largura de banda maior comparadas à mem global
- Usadas quando variáveis são acessadas com frequência pelo *kernel*
- **Alocação estática** usa ***__shared__*** como modificador na declaração da variável
- **Alocação dinâmica**
  - Usa-se ***extern __shared__*** no *kernel*
  - Invocação do *kernel* no *host* usa um parâmetro a mais (nr de bytes)
- A memória compartilhada não permite transferência de dados com o *host*
  - A memória global deve ser usada para as transferências de/para o *host*
- ***Exemplo 02 – ilustra alocação dinâmica***

Memória Compartilhada

- Exemplo: ***vetorC_dev[ ]*** é definido na memória compartilhada dentro do *kernel launch* do *kernel* no *host* informa o total de bytes da memória compartilhada

```
...
__global__ void soma(int *vetorA, int *vetorB, int *vetorC, int tam){
   //Declara o vetorC _dev na memória compartilhada, alocado dinamicamente
  // tam aqui determina o nr itens do vetor, nao de bytes

   extern __shared__ int vetorC_dev[ ];
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   ...


int main(int argc,char **argv){
...
 soma <<<blocksPerGrid,threadsPerBlock,tam*sizeof(int)>>> (A_d,B_d,C_d,tam);
...
```

Memória Constante

- Permite apenas leitura no *kernel*
- Tem um menor tempo de latência que a memória global
- São visíveis para todas as *threads* da grade e o tempo de vida é da aplicação
- Memórias cache por SM otimizam o uso da memória constante
- Para declarar deve-se usar o modificador ***__constant__*** no *host*; não no *device*
  - ***__constant__ int naomuda = 100***
- O *device* não pode mudar o valor da constante, mas o *host* pode, usando
  - **cudaMemcpyToSymbol( )**

```
cudaError_t cudaMemcpyToSymbol (
        const char *     symbol,
        const void *     src,
        size_t       count,
        size_t       offset,   /*opcional */
        enum cudaMemcpyKind  kind   /*opcional */       );
```

- ***Exemplo 03 – ilustra o uso da memória constante***

Memória Unificada

- Permite a cópia implícita de memória entre *host* e *device*
  - Não reduz o tempo de execução de um programa
  - As transferências de memória continuam acontecendo, sob demanda, transparentemente
- Simplifica a programação CUDA e elimina o uso da *CudaMemcpy( )*

- Questão de terminologia: memória unificada é o mesmo que gerenciada (***managed***)

- Para alocar variáveis **estaticamente**
  - Variável global no host usando ***__managed__***
- Para alocar variáveis **dinamicamente**
  - Usar ***cudaMallocManaged()*** no *host*

*cudaError_t cudaMallocManaged (const char ** ptr, size_t size, unsigned flag)*

- *flag* ***cudaMemAttachGlobal*** (esta é a flag padrão, não precisa indicar)
  - Memória alocada é acessível em qualquer *kernel* em execução

- *flag* ***cudaMemAttachHost***
  - Memória alocada só é acessível nos *kernels* lançados pela *thread* que fez a alocação

- ***Exemplo 04 – ilustra o uso da memória unificada***

Impacto no Desempenho dos Tipos de Memória

- Tempos de resposta dos diferentes tipos de memória
- Execuções no cluster do LaSDPC

GeForce GTX650 (Kepler, GDDR5, 1GB, 384 cuda cores). Ubuntu 18.04.		
Nome	Tempo de execução do *kernel* (ms)	Tempo de execução total (ms)
soma_vet_global	10,019	583,933
soma_vet_managed	74,209	697,400
soma_vet_const	Nulo	nulo
soma_vet_shared	0,006	533,000
soma_vet_local	8,259	262,233

Tabela 1: Soma de três vetores de 50.000.000 de inteiros em Cuda. (GTX650)

Barreiras em CUDA

- Há duas barreiras principais
- ***__synthreads()***
  - executada no *kernel,* força sincronização das *threads* de um mesmo bloco
- ***cudaDeviceSynchronize()***
  - Executada no *host*, força a sincronização de todas as *threads* do *device*
  - Lançamento de um *kernel* não é síncrono, i.e., a próxima instrução no *host*, após o lançamento, pode ser executada antes do *kernel* terminar na GPU
  - Se a próxima instrução no *host* solicitar execução na GPU, tais execuções são serializadas em uma *stream* de execução

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

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

__global__ void hello(){
   printf("Hello ");
}

int main(int argc,char **argv){
   hello<<<1,1>>>();
   cudaDeviceSynchronize();
   printf("World\n");
}
```

Exemplo 05 ilustra uso de barreiras
Exemplo 06 ilustra uso de barreiras e mem comp estática/dinâmica

Exemplos

- Exemplo 01 – Soma matrizes quadradas usando memória global
- Exemplo 02 – Soma dois vetores com alocação dinâmica de memória compartilhada
- Exemplo 03 – Multiplica vetor por valor escalar usando memória constante
- Exemplo 04 – Incrementa e decrementa valores em um vetor, usando memória compartilhada definida estática e dinamicamente
- Exemplo 05 – Troca valores de um vetor em suas posições (exemplo __*syncthreads()*)
- Exemplo 06 – Soma vetores usando memória unificada
- Exemplo 07 – Multiplicação de matrizes usando diversos recursos aprendidos na aula

Referências

Barlas, G. (2014). Multicore and GPU Programming: An integrated approach. Elsevier. Capítulo 6.

Rauber, T., & Rünger, G. (2013). Parallel Programming. Springer. Second edition. Capítulo 7.

Patterson, D.A., Hennessy, J.L. Computer organization and design : the hardware/software interface, 5th ed., Elsevier, Amsterdam, 2014,

Kirk, D.B., Hwu, W.W., Programming Massively Parallel Processors: a hands-on approach. 2nd ed., Morgan Kaufman, NVIDIA, 2013.

NVIDIA-PG (2019), Cuda C Programming: design guide, PG-02829-001_v10.1, May, 2019, Cap 01 (Introduction) e Cap 02 (Programming Model).

CUDA RUNTIME API, Api Reference Manual, NVIDIA, July 2019.

Sanders, J., & Kandrot, E. (2010). CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional.

Memória e Sincronização em CUDA

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP