Operações Atômicas e Streams 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

Operações Atômicas em CUDA

- Operações atômicas protegem regiões críticas em CUDA
  - *Capability* > 2.0 suportam essas operações
    - ***atomicAdd()*** é um exemplo de operação atômica em função ***__device__***
  - *Capability* > 6.0 suportam novos tipos de operações atômicas
    - ***atomicAdd_system()*** garante atomicidade também em relação à CPU e outras GPUs
    - ***atomicAdd_block()*** garante atomicidade para *threads* de um mesmo bloco

- Todas as funções atômicas têm ponteiro para endereço de memória compartilhado pela variável entre as threads e outros argumentos da operação atômica.

- Operações atômicas podem ser:

Aritméticas	Soma	*type atomicAdd(type* address,type val)*
	Subtração	*type atomicSub(type* address,type val)*
	Substituição	*type atomicExch(type* address,type val)*
	Mínimo	*type atomicMin(type* address,type val)*
	Máximo	*type atomicMax(type* address,type val)*
	Incr. Cond	*unsigned int atomicInc(unsigned int* address, unsigned int val)* *((old >= val) ? 0 : (old+1))*
	Decr. Cond	*unsigned int atomicDec(unsigned int* address, unsigned int val)* *(((old == 0) || (old > val)) ? val : (old-1))*
	Subst. Cond	*unsigned type atomicCAS (type* address, type compare, type val)* *compare & swap => old == compare ? val : old*

Operações Atômicas em CUDA

- **Lógicas binárias**

AND	*type atomicAnd(type* address,type val)*
OR	*type atomicOr(type* address,type val)*
XOR	*type atomicXor(type* address,type val)*

- Exemplo ***00-soma-elems-vet-atomic***
  - Soma os elementos de um vetor usando operações atômicas

Memória Não Paginável em CUDA

- Execuções assíncronas permitem *sobrepor* execuções na GPU (a partir da *Capability* 2.0)
  - Transferência de/para memória, lançamentos de *kernels* e alocações de memória
  - Permitem diminuir *overhead* e esconder a latência de memória.

- CUDA usa primitivas ***streams*** que determinam uma sequência de comandos na GPU
  - Múltiplos *streams* podem executar concorrentemente, maximizando o uso de recursos

- Precisam utilizar memórias não paginàveis (***pinned memories*** ou *page-locked*) **no *host***
  - Isso habilita a execução concorrente das atividades de uma *stream*

cudaError_t cudaMallocHost(void **ptr, size_t size)

*cudaFreeHost(void *ptr)*

Memória Mapeada em CUDA

- Há também a opção de memória mapeada (*mapped memory* ou *zero-copy memory*)
  - Permite memória não paginável do host ser mapeado na memória do device
    - Mesmo ponteiro pode ser usado tanto no host quanto no device
      - Endereço Virtual Unificado (*Unified Virtual Address* – UVA)

cudaError_t cudaHostAlloc(void **ptr, size_t size, unsigned int flag)

Onde ***flag*** pode ser:

cudaHostAllocDefault: função comporta-se exatamente como a ***cudaMallocHost()***

cudaHostAllocPortable: memória alocada é considerada não paginável para todos os contextos CUDA, não somente aquele que fez a alocação

cudaHostAllocMapped: alocação é feita no *device*. Aqui pode-se usar um único ponteiro para acessar a memória do *host* ou *device*. Pode-se atribuir um ponteiro específico para acessar a memória do *device* por meio da função ***cudaHostGetDevicePointer()***

cudaHostAllocWriteCombined: Aloca a memória como escrita combinada (*WC - write combined*). A memória com WC não vai para *cache* L1 e L2, liberando espaço destas para outras memórias. Também não requer *snoopy* (coerência de cache) e assim pode ser transferida através do barramento PCI Express mais rapidamente (até 40%)

Resumo das Alocações e Cópias de Memória

- Um resumo das alocações e cópias de memória vistas até agora:
  - ***malloc()*** => aloca memória paginável no *host*
  - ***cudaMalloc()*** => aloca memória no *device*
  - ***cudaMallocManaged()*** => aloca memória paginável com ptr para ambos: *host* e *device*
    memória unificada ou gerenciada (managed memory)
    __managed__
  - ***cudaMallocHost()*** *=> aloca memória não paginável no host (pinned memory)*
  - ***cudaHostAlloc()*** => aloca memória não paginável ou só no *host* ou em ambos
    memória mapeada (*mapped memory*)

  - ***free()*** => libera memória paginável alocada no *host*
  - ***cudaFree()*** => libera memória alocada no *device* (usada com *cudaMallocManaged()*)
  - ***cudaFreeHost()*** => libera memória não paginável no *host*
    usada com *cudaMallocHost()* ou *cudaHostAlloc()*

  - ***cudaMemcpy()*** => faz cópia da memória entre *host* e *device* de maneira síncrona
  - ***cudaMemcpyAsync()*** => faz cópia assíncrona entre *host* e *device*
    requer *pinned memory*

Streams de Execuções em CUDA

- *Streams* são sequências ordenadas de comandos CUDA para execução sequencial
- Múltiplas *streams* podem executar concorrentemente nos recursos disponíveis
  - Representadas pelo tipo ***cudaStream_t***

- Criação de uma *stream*

  cudaError_t cudaStreamCreate(cudaStream_t *pStream)

- Destruir uma stream

  cudaError_t cudaStreamDestroy(cudaStream_t stream)

- Para inserir commandos CUDA na *stream*

cudaError_t cudaMemcpyAsync(void *dst, void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream)

Análoga à ***cudaMemcpy()***, porém, faz a cópia de maneira assíncrona.
Especifica a ***stream*** onde será executada.

- Outra maneira de inserir commandos na *stream* é invocando um novo *kernel*

  Kernel <<<gridDim, blockDim, sharedMemory, stream>>> (args)

- Para sincronizar as *threads* de uma *stream*

  cudaError_t cudaStreamSynchronize(cudaStream_t stream)

Exemplos

- Exemplo ***01-soma-vet-pinned*** => Faz a soma dos elementos de dois vetores
  - Exemplifica *cudaMallocHost()* para alocar memoria paginada no *host* e *cudaFreeHost()* para desalocar

- Exemplo ***02-soma-vet-mapped*** => Faz a soma dos elementos de dois vetores
  - Exemplifica memoria mapeada com *cudaHostAlloc()* usando parâmetro *cudaHostAllocMapped* para alocar memoria tanto no host quanto no device.
  - Copias entre host e device são implícitas, igual aa memoria unificada.
  - Usa *cudaDeviceSynchronize()* antes da impressão do resultado para corretude

- Exemplo ***03-soma-vet-stream*** => Faz a soma dos elementos de dois vetores
  - Exemplifica inserções em uma *stream, cudaMallocHost* para alocar mem não paginável no host e faz cópia assíncrona com *cudaMemcpyAsync()*.
  - Usa *cudaStreamSynchronize()* para aguardar toda a *stream* terminar.
  - O algoritmo divide "***tam***" elems por "***streams_nr***" e encontra "***threadsPerGrid***" e "***blocksPerGrid***“. O vetor no *device* tem o tamanho de ***threadsPerGrid***.

- Exemplo ***04-soma-vet-stream-2*** => Faz a soma dos elementos de dois vetores
  - Exemplifica uso de duas *streams* diferentes (1 e 2)

- Exemplo ***05-streams_MIMD*** => Faz a soma dos elementos de dois vetores em uma *stream* e em outra *stream* faz a multiplicação de um vetor por um valor escalar
  - Exemplifica computações diferentes em *streams* concorrentes diferentes

Exemplos

- Exemplos para a soma de vetores:
  - Esquema para a divisão do vetor de dados sobre grades, blocos e threads
  - Nos exemplos, as grades são inseridas sequencialmente na mesma *stream* ou em *streams* diferentes

	Grade/Stream 1								Grade/Stream 2							
Blocos	Blc 0		Blc 1		Blc 2		Blc 3		Blc 0		Blc 1		Blc 2		Blc 3	
Threads	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Dados	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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.

Operações Atômicas e *Streams* 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