

Parallel Scientific Computation

CUDA 2

J.-H. Parq

IPCST

Seoul National University

SIMT

- 32 threads (warpSize) is the maximum number of simultaneously executable threads per scheduler.
- Unlike SIMD, SIMT instructions **can specify** the execution and branching behavior of a single thread.
- But specific execution or branches **reduce** efficiency.

```
– Ex.) ID = threadIdx.x;  
      if (ID>15) {  
          a = a + ID;  
      }  
      else {  
          a = a - ID;  
      }
```

➤ Threads 0~15 act after threads 16~31 finish.

Restrictions of GPU Computation

1. Performance speed depends on the way of memory access (including cache usage).
 - CPU memory access also has such rules but is relatively less restrictive.
2. Different instructions in a warp are possible but increase the total execution time.
 - Cf.) SIMD, OpenMP
3. Data transfer between a GPU device and its host is usually slow.

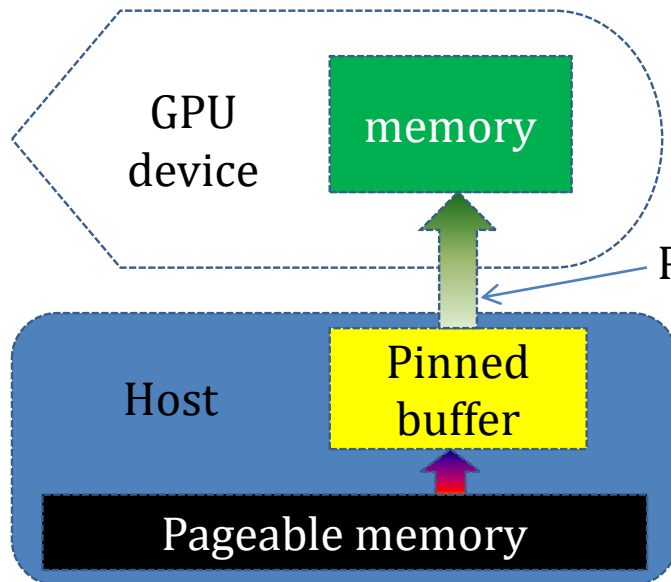
Page-Locked Host Memory

- (a. k. a.) Pinned memory
 - Enabling asynchronous data transfer
 - ‘`cudaHostAlloc`’ instead of ‘`malloc`’
 - There are other allocating functions. See the document ‘CUDA Runtime API’
 - ‘`cudaFreeHost`’ instead of ‘`free`’
 - The amount of page-locked memory is usually **limited by OS** because pinned memories are unable to extend to virtual memories and reduces the whole system performance.

Asynchronous Transfer

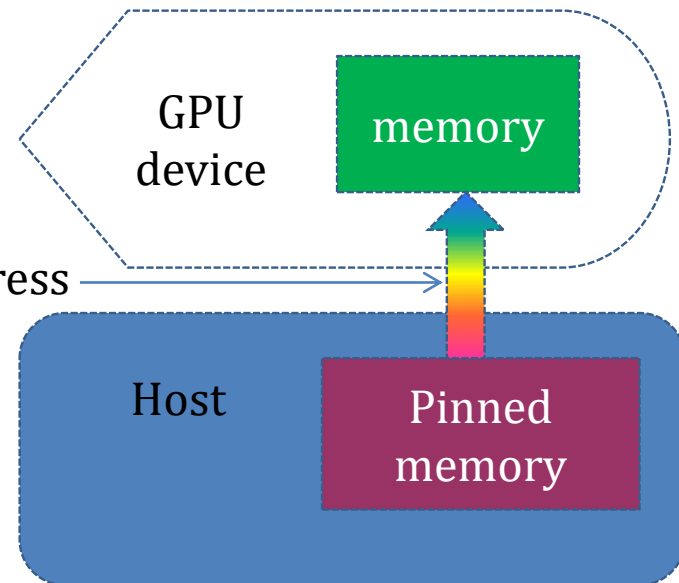
Pageable memory

- Ordinarily allocated
- Concurrent data transfer



Page-locked memory

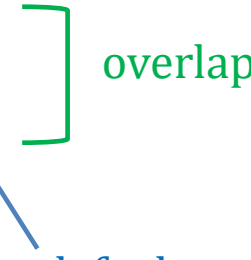
- Used for asynchronous transfer



Asynchronous Transfer (CUDA C)

- Form 1)

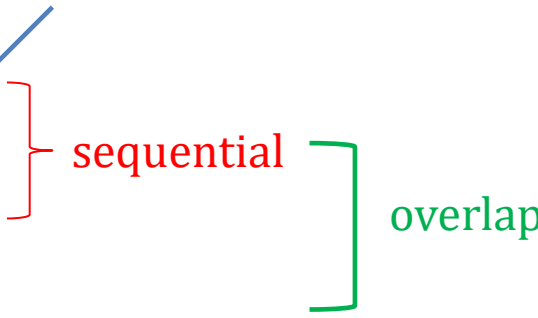
```
cudaMemcpyAsync(d_a, h_a, size,  
    cudaMemcpyHostToDevice, 0);  
cpuFunction();
```



overlap

- Form 2)

```
cudaMemcpyAsync(d_a, h_a, size,  
    cudaMemcpyHostToDevice, 0);  
kernel<<<grid, block>>>(a_d);  
cpuFunction();
```



default stream

sequential

overlap

Asynchronous Transfer (CUDA C)

- Form 3)

```
cudaStream_t stream1, stream2;
```

```
.....
```

```
cudaHostAlloc(&h_v, size, cudaHostAllocDefault)
```

```
cudaStreamCreate(&stream1); cudaStreamCreate(&stream2);
```

```
.....
```

```
cudaMemcpyAsync(d_v, h_v, size, cudaMemcpyHostToDevice, stream1);
```

```
kernel<<<grid, block, 0, stream2>>>(d_a);
```

```
.....
```

```
cudaStreamDestroy(stream1); cudaStreamDestroy(stream2);
```

```
cudaFreeHost(h_v);
```

Asynchronous Transfer (CUF)

```
Integer (kind=cuda_stream_kind) :: stream1, stream2
```

```
Real, Pinned, Allocatable :: h_v(:)
```

```
Allocate(h_v(N), STAT=istat, PINNED=pinnedFlag)
```

```
.....
```

```
istat = cudaStreamCreate(stream1)
```

```
istat = cudaStreamCreate(stream2)
```

```
istat = cudaMemcpyAsync(d_v, h_v, N, stream1)
```

```
Call kernel<<<grid, block, 0, stream2>>>(d_a)
```

```
.....
```

```
istat = cudaStreamDestroy(stream1)
```

```
istat = cudaStreamDestroy(stream2)
```

```
Deallocate(a)
```


Asynchronous Transfer (OpenACC)

- ‘**async**’ clause
 - To ‘**kernels**’, ‘**parallel**’ or ‘**update**’
 - ‘**wait**’ directive: similar to the ‘wait’ in MPI

!\$acc parallel loop async

Do i=1,N

c(i) = a(i)/b(i)

End do

!\$acc update self(c) async

!\$acc wait

#pragma acc parallel loop async(1)
for (int i=0; i<N; i++) a[i] = i%8;

#pragma acc parallel loop async(2)
for (int i=0; i<N; i++) b[i] = 2*a[i];

#pragma acc wait(1) async(2)

.....

Events

- Exact way to measure performance time

CUDA C

```
cudaEvent_t start, end;  
float elapsed;  
cudaEventCreate(&start);  
cudaEventCreate(&end);  
cudaEventRecord(start, stream);  
.....  
cudaEventRecord(end, stream);  
cudaEventSynchronize(end);  
cudaEventElapsedTime(&elapsed,  
                    start, end);
```

CUDA Fortran

```
Type (cudaEvent) :: start, end  
Real :: time  
istat = cudaEventCreate(start)  
istat = cudaEventCreate(end)  
istat = cudaEventRecord(start, 0)  
.....  
istat = cudaEventRecord(end, 0)  
istat = cudaEventSynchronize(end)  
istat = cudaEventElapsedTime( &  
                             time, start, end)
```

cudaEventDestroy

Error Handling

- CUDA C examples

1. `if (cudaMalloc(&d_a, size) != cudaSuccess) printf("Error!");`
2. `Kernel<<M, N>>(d_a, d_b);`
`cudaDeviceSynchronize();`
`cudaError_t error = cudaGetLastError();`
`if (error != cudaSuccess) {`
 `printf("CUDA error: %s\n", cudaGetErrorString(error));`
 `exit(1); }`

– You can make a wrapper and use it.

```
#define CUDA_CHECK(call) \  
    if((call) != cudaSuccess) { \  
        .....
```

Error Handling

- CUDA Fortran examples

1. `istat = cudaEventSynchronize(end)`

`If (istat /= cudaSuccess) Then`

`Write(*,*) cudaGetErrorString(istat)`

2. `Call kernel<<M, N>>(d_a, d_b)`

`ierrS = cudaGetLastError()`

`ierrA = cudaDeviceSynchronize()`

`If (ierrS /= cudaSuccess) write(*,*) &`

`'Sync kernel error:', cudaGetErrorString(ierrS)`

`If (ierrA /= cudaSuccess) write(*,*) &`

`'Async kernel error:', cudaGetErrorString(ierrA)`

Matrix Multiplication (C)

- Simple version

```
__global__ void simpleMultiply(float *a, float* b, float *c, int M, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    double sum = 0.0;
    for (int i = 0; i < M; i++) sum += a[row*M+i] * b[i*N+col];
    c[row*N+col] = sum;
}
```

Matrix Multiplication (C)

- Using shared memory

```
__global__ void sharedABMultiply(float *a, float* b, float *c, int L, int M, int N)
{
    __shared__ double aTile[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ double bTile[BLOCK_SIZE][BLOCK_SIZE];
    int brow = blockIdx.y;
    int bcol = blockIdx.x;
    int trow = threadIdx.y;
    int tcol = threadIdx.x;
    int row = brow*blockDim.y+trow;
    int col = bcol*blockDim.x+tcol;
    double sum = 0.0;
```

See 'CUDA C Programming Guide' for more details

Matrix Multiplication (C)

- Using shared memory

```
nBlock = L/BLOCK_SIZE + (L%BLOCK_SIZE != 0);
for (k = 0; k < nBlock; k++) {
    aTile[trow][tcol] = a[(BLOCK_SIZE*brow+trow)*M+k*BLOCK_SIZE+tcol];
    bTile[trow][tcol] = b[(k*BLOCK_SIZE+trow)*N+BLOCK_SIZE*bcol+tcol];
    __syncthreads();
    for (int i = 0; i < BLOCK_SIZE; i++) sum += aTile[trow][i] * bTile[i][tcol];
    __syncthreads();
}
if (row < L && col < N) c[(row)*N+col] = sum;
}
```

See 'CUDA C Programming Guide' for more details

Matrix Multiplication (CUF)

```
attributes(global) subroutine mmul_kernel( A, B, C, N, M, L )  
  real :: A(N,M), B(M,L), C(N,L)  
  integer, value :: N, M, L  
  integer :: i, j, kb, k, tx, ty  
  real, shared :: Asub(16,16), Bsub(16,16)  
  real :: Cij  
  
  tx = threadidx%x  
  ty = threadidx%y  
  ! This thread computes  $C(i,j) = \text{sum}(A(i,:) * B(:,j))$   
  i = (blockidx%x-1) * 16 + tx  
  j = (blockidx%y-1) * 16 + ty  
  Cij = 0.0
```

See 'CUDA Fortran Programming Guide' for more details

Matrix Multiplication (CUF)

! Do the k loop in chunks of 16, the block size

do kb = 1, M, 16

 Asub(tx,ty) = A(i,kb+ty-1)

 Bsub(tx,ty) = B(kb+tx-1,j)

 call syncthreads()

 do k = 1,16

 Cij = Cij + Asub(tx,k) * Bsub(k,ty)

 enddo

 call syncthreads()

enddo

C(i,j) = Cij

end subroutine mmul_kernel

See 'CUDA Fortran Programming Guide' for more details

2-D Array in CUDA C?

```
float A[N][N], x[N], y[N];  
float (*d_A)[N], *d_x, *d_y;  
.....  
cudaMalloc((void**)&d_A, (N*N)*sizeof(float));  
cudaMalloc((void**)&d_x, N*sizeof(float));  
cudaMalloc((void**)&d_y, N*sizeof(float));  
cudaMemcpy(d_A, A, (N*N)*sizeof(float), cudaMemcpyHostToDevice);  
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);  
.....  
dim3 threadsPerBlock(16, 16);  
dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);  
MatVec<<<numBlocks, threadsPerBlock>>>(d_A, d_x, d_y);  
cudaMemcpy(y, (d_y), N*sizeof(float), cudaMemcpyDeviceToHost);  
.....
```

CUDA Pitched Memory (C)

- Conventional dynamic 2D array



- What if the CUDA array is 1D while the host array is 2D or 3D? Use the below functions.
 - Allocation: `cudaMallocPitch` / `cudaMalloc3D`
 - Transfer: `cudaMemcpyToSymbol` / `cudaMemcpy2D` / `cudaMemcpyFromSymbol` / etc.
- See 'CUDA C Programming Guide' or 'CUDA Runtime API' for more information.

Multiple GPUs

- Checking the number of GPU devices
 - `cudaGetDeviceCount(int *count)`
- Properties of a GPU device
 - `cudaGetDeviceProperties(cudaDeviceProp *prop, int devNum)`
 - Ex.) `cudaDeviceProp prop;`
`cudaGetDeviceProperties(&prop, 1);`
- Entering the mode of a GPU device
 - `cudaSetDevice(int devNum)`
 - Ex.) `cudaSetDevice(0)` → using GPU 0 only

Multiple GPUs

- Access to another GPU's memory
 - `cudaDeviceCanAccessPeer(*can, devN, peerN)`
 - `cudaDeviceEnablePeerAccess(peerN, 0)`
 - `cudaDeviceDisablePeerAccess(peerN)`
- Memory copy to another GPU device
 - `cudaMemcpyPeer(dst, dstDev, src, srcDev, count)`
 - Count = size(byte) (c) or number of elements (CUF)
 - Bytes if dst and src are of type(C_DEVPTR) in CUF
 - `cudaMemcpyPeerAsync(dst, dstDev, src, srcDev, count, stream)`

Other Kinds of Data Transfer

- An example of setting a constant variable

```
int h_c = 256;
```

```
__constant__ int c;
```

```
cudaMemcpyToSymbol("c", &h_c, sizeof(int));
```

- Zero copy
 - Mapped pinned memory is necessary.
 - No explicit data transfer

Other Kinds of Data Transfer

- Zero-copy example
 - from ‘CUDA C Best Practices’

```
float *a_h, *a_map;
```

```
...
```

```
cudaGetDeviceProperties(&prop, 0);
```

```
if (!prop.canMapHostMemory)
```

```
    exit(0);
```

```
cudaSetDeviceFlags(cudaDeviceMapHost);
```

```
cudaHostAlloc(&a_h, nBytes, cudaHostAllocMapped);
```

```
cudaHostGetDevicePointer(&a_map, a_h, 0);
```

```
kernel<<<gridSize, blockSize>>>(a_map);
```

❖ In case of CUF, see ‘CUDA Fortran Programming Guide’ 5.2

Other Kinds of Memory

- These memory types can be allocated by using `cudaHostAlloc` with the other options instead of `cudaHostAllocDefault`.
 - Mapped pinned: `cudaHostAllocMapped`
 - Zero copy requires this.
 - If this memory is not read from CPU, you can replace it with '`cudaHostAllocWriteCombined`' to enhance access from GPU.
 - Portable pinned: `cudaHostAllocPortable`
 - This enables zero copy between GPUs. See the book 'CUDA by Example' for details.

Unified Virtual Address Space

- For 64-bit processes and GPUs of compute capability 2.0 or higher
- This property **simplifies** the **zero copy**.
 - Without this, the host variable and the device variable for the same data are different.
 - You can use the same variable for the same data.
 - You don't need `cudaHostGetDevicePointer()` anymore.
- See 'CUDA C Programming Guide' for more information.
 - Cf.) Also, find information about 'unified memory'.
 - Requirement: Kepler or newer GPU. 64-bit Linux or Windows.

Atomic Functions

- Used in a kernel (global or shared memory)
 - Currently, only one atomic function in CUDA C for double-precision floating point numbers (compute capability 6.x or higher)
 - `y = atomicAdd(&x[0], c);` $\rightarrow y = x[0]; x[0] += c;$
 - For the other atomic functions (ex. integer functions), see 'CUDA C Programming guide'.
 - In CUF, `atomicadd`, `atomicsub`, `atomicmax`, `atomicmin`, and `atomicexch` are available for `real(8)` type numbers. (compute capability 2.x or higher)
 - `y = atomicexch(x, c);` $\rightarrow y = x \ \& \ x = c$
 - You can find more information in 'CUDA Fortran Programming Guide'.

CUDA Dynamic Parallelism

- Launching another kernel (child kernel) from a kernel (parent kernel)
 - Cf.) nested parallelism in OpenMP
 - Requiring compute capability 3.5 or higher
 - Advantage: more flexible programming
 - Easier to reduce data transfer and to balance loads
 - Disadvantage: kernel launching overheads
- ❖ See the 'CUDA C Programming guide' for more information.

Reduction Example (OpenACC)

Fortran

!\$acc parallel loop collapse(2)

Do i=1,M

Do j=1,N

sum = 0d0

!\$acc loop reduction(+:sum)

Do k=1,L

sum = sum + A(i,k)*B(k,j)

End do

C(i,j) = sum

End do

End do

C

#pragma acc parallel loop collapse(2)

for (i = 0; i < M; i++) {

for (j = 0; j < N; j++) {

double cij = 0.0;

#pragma acc loop reduction(+:cij)

for (k = 0; k < L; k++) {

cij += A[i][k]*B[k][j];

}

C[i][j] = cij;

}

}

Reduction Example (CUDA C)

- Summation on shared memory

- ID = thread ID

- half = blockDim.x/2;

- while (half >= 1) {

- if (ID < half) psum[ID] += psum[ID+half];

- __syncthreads();

- half /= 2;

- }

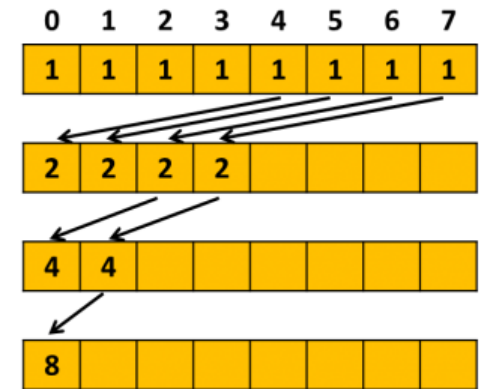
- Free of bank conflict

- See the 'CUDA Handbook' for details or reduction with atomic operations.

- But you can optimize this further.

- First add during load, unrolling, multiple adds

- See the reference (Harris or Owens) for details



(adding before loading to shared memory)

Figure from NVIDIA homepage

Warp Functions

- You can enhance the performance of reduction by using the warp reduce functions (compute capability 8.x or higher) or the warp shuffle functions (compute capability 3.x or higher)
- See 'CUDA C Programming Guide B.21 & B.22' for further information.

CUDA Libraries

- NVIDIA libraries
 - CUDA Math Library
 - cuBLAS, cuSolver, cuSPARSE, AmgX
 - cuFFT
 - cuRAND
 - cuTENSOR
 - Thrust
 - NVSHMEM, NCCL
 - Libraries for deep learning: cuDNN, TensorRT,

CUDA Libraries

- Non-NVIDIA libraries
 - ArrayFire
 - MAGMA (Matrix Algebra on GPU and Multicore Architectures)
 - IMSL Fortran Numerical Library
 - Gunrock
 - Graph processing
 - Triton Ocean SDK
 - CHOLMOD
 - Sparse direct solvers

Information Leakage

- CUDA runtime accepts any processing requests to a GPU and queued them.
 - A vicious process can catch the last data (stored in **shared memory** and **global memory**) of the previous process which used the GPU.
 - **Countermeasure**: clean GPU shared memory and global memory at the end of your process if security matters.
- Ref: R. Di Pietro *et al.*, “CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix”, ACM Transactions on Embedded Computing Systems, Vol. 15, No. 1, Article 15 (2016).

References

- CUDA C Programming Guide
- CUDA C Best Practices
- CUDA Fortran Programming Guide
- Introduction to CUDA Fortran
 - GPU Technology Conference 2013

References

- OpenACC homepage
<https://www.openacc.org/>
- M. Harris, “Optimizing Parallel Reduction in CUDA”
- J. Owens, CS 223 Guest Lecture “GPU Reduce, Scan, and Sort”

References

- J. Sanders & E. Kandrot, CUDA by Example
- N. Wilt, The CUDA Handbook
- S. Cook, CUDA Programming