Introduction to CUDA Parallel Programming
CUDA平行計算導論

https://ceiba.ntu.edu.tw/1092Phys8061_CUDA

Professor Ting-Wai Chiu (趙挺偉)
Email: twchiu@phys.ntu.edu.tw
Physics Department
National Taiwan University

This lecture will cover:

- CUDA Libraries: cuBLAS, cuFFT,…
- cuBLAS – Saxpy: $|B\rangle \leftarrow \alpha|A\rangle + |B\rangle$
- cuBLAS – Sgemm: $C \leftarrow \alpha A \cdot B + \beta C$

The cuBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime. It allows the user to access the computational resources of NVIDIA GPU.
https://docs.nvidia.com/cuda/cublas/index.html

BLAS – Basic Linear Algebra Subroutines

http://www.netlib.org/blas/

BLAS Routines

LEVEL 1 **SINGLE**

- SROTG - setup Givens rotation
- SROTMG - setup modified Givens rotation
- SROT - apply Givens rotation
- SROTM - apply modified Givens rotation
- SSWAP - swap x and y
- SSCAL - x = a*x
- SCOPY - copy x into y
- SAXPY - y = a*x + y
- SDOT - dot product
- SDSDOT - dot product with extended precision accumulation
- SNRM2 - Euclidean norm
- SCNRM2- Euclidean norm
- SASUM - sum of absolute values
- ISAMAX - index of max abs value

BLAS – Basic Linear Algebra Subroutines

http://www.netlib.org/blas/

BLAS Routines

LEVEL 1 **DOUBLE**

DROTG - setup Givens rotation
DROTMG - setup modified Givens rotation
DROT - apply Givens rotation
DROTM - apply modified Givens rotation
DSWAP - swap x and y
DSCAL - x = a*x
DCOPY - copy x into y
DAXPY - y = a*x + y
DDOT - dot product
DSDOT - dot product with extended precision accumulation
DNRM2 - Euclidean norm
DZNRM2 - Euclidean norm
DASUM - sum of absolute values
IDAMAX - index of max abs value

BLAS – Basic Linear Algebra Subroutines

http://www.netlib.org/blas/

BLAS Routines

LEVEL 1 **COMPLEX**

CROTG - setup Givens rotation
CSROT - apply Givens rotation
CSWAP - swap x and y
CSCAL - x = a*x
CSSCAL - x = a*x
CCOPY - copy x into y
CAXPY - y = a*x + y
CDOTU - dot product
CDOTC - dot product, conjugating the first vector
SCASUM - sum of absolute values
ICAMAX - index of max abs value

BLAS – Basic Linear Algebra Subroutines

http://www.netlib.org/blas/

BLAS Routines

LEVEL 1 **DOUBLE COMLPEX**

ZROTG - setup Givens rotation
ZDROTF - apply Givens rotation
ZSWAP - swap x and y
ZSCAL - x = a*x
ZDSCAL - x = a*x
ZCOPY - copy x into y
ZAXPY - y = a*x + y
ZDOTU - dot product
ZDOTC - dot product, conjugating the first vector
DZASUM - sum of absolute values
IZAMAX - index of max abs value

BLAS – Basic Linear Algebra Subroutines

http://www.netlib.org/blas/

BLAS Routines

LEVEL 2   S(Single) / D(Double) / C(Complex) / Z(Double Complex)

SGEMV **- matrix vector** multiply
SGBMV - banded matrix vector multiply
SSYMV - symmetric matrix vector multiply
SSBMV - symmetric banded matrix vector multiply
SSPMV - symmetric packed matrix vector multiply
STRMV - triangular matrix vector multiply
STBMV - triangular banded matrix vector multiply
STPMV - triangular packed matrix vector multiply
STRSV - solving triangular matrix problems
STBSV - solving triangular banded matrix problems
STPSV - solving triangular packed matrix problems
SGER - performs the rank 1 operation A := alpha*x*y' + A
SSYR - performs the symmetric rank 1 operation A := alpha*x*x' + A
SSPR - symmetric packed rank 1 operation A := alpha*x*x' + A
SSYR2 - performs the symmetric rank 2 operation, A := alpha*x*y' + alpha*y*x' + A
SSPR2 - performs the symmetric packed rank 2 operation, A := alpha*x*y' + alpha*y*x' + A

BLAS – Basic Linear Algebra Subroutines

http://www.netlib.org/blas/

BLAS Routines

Level 3 **Z(Double Complex) / C(Complex) / D(Double) / S(Single)**

ZGEMM - **matrix matrix** multiply
ZSYMM - symmetric matrix matrix multiply
ZHEMM - hermitian matrix matrix multiply
ZSYRK - symmetric rank-k update to a matrix
ZHERK - hermitian rank-k update to a matrix
ZSYR2K - symmetric rank-2k update to a matrix
ZHER2K - hermitian rank-2k update to a matrix
ZTRMM - triangular matrix matrix multiply
ZTRSM - solving triangular matrix with multiple right hand sides

cuBLAS

The cuBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime. It allows the user to access the computational resources of NVIDIA GPU.
https://docs.nvidia.com/cuda/cublas/index.html

BLAS Sgemm $C \leftarrow \alpha A \cdot B + \beta C$

cuBLAS cublasSgemm for single GPU

cuBLAS cublasXtSgemm for multi-GPUs
(Note that only some routines are available for multi-GPUs)

cuBLAS API

The cuBLAS Library exposes three sets of API:

- The cuBLAS API, which is simply called cuBLAS API (starting with CUDA 6.0)
- The CUBLASXT API (starting with CUDA 6.0)
- The cuBLASLt API (starting with CUDA 10.1)

To use the cuBLAS API, the application must allocate the required matrices and vectors in the GPU memory space, fill them with data, call the sequence of desired cuBLAS functions, and then upload the results from the GPU memory space back to the host. The cuBLAS API also provides helper functions for writing and retrieving data from the GPU.

To use the CUBLASXT API, the application must keep the data on the Host and the Library will take care of dispatching the operation to one or multiple GPUs present in the system, depending on the user request.
If you can find your needed routine in CUBLASXT API, then use it rather than the corresponding one in cuBLAS API.

The cuBLASLt API is a lightweight library dedicated to GEneral Matrix-to-matrix Multiply (GEMM) operations with a new flexible API.

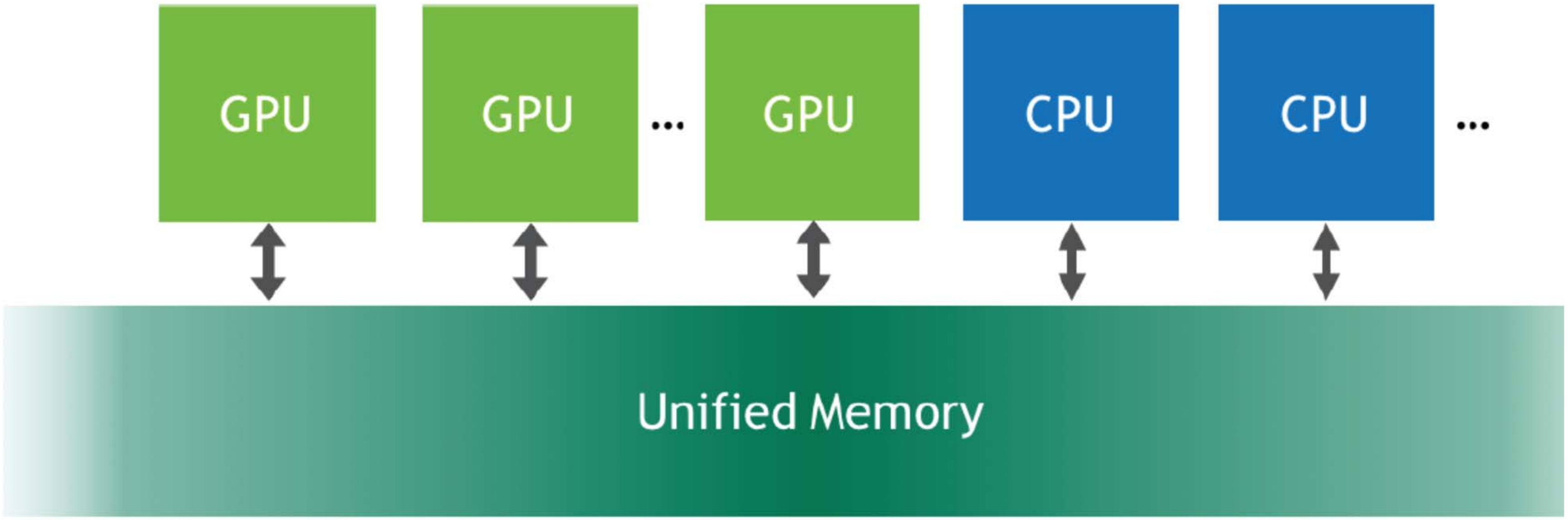
SAXPY(y ← a*x + y), simple kernel with 1-GPU

```
__global__ void Saxpy(const float alpha, const float* A, float* B, long N)
{

    long i = blockDim.x * blockIdx.x + threadIdx.x;

    while (i < N) {
        B[i] += alpha*A[i];
        i += blockDim.x * gridDim.x;  // go to the next grid
    }
    __syncthreads();
}

// complete code in twqcd80:/home/cuda_lecture_2021/Saxpy_1GPU/Saxpy_1gpu.cu
```

SAXPY (y ← a*x + y), cuBLAS with 1-GPU

```
  #include "cublas_v2.h"          // header for CUBLAS

  int main(void){
  ...
    cublasHandle_t handle;          // CUBLAS context
  ...
    cublasCreate(&handle);       // initialize CUBLAS context
    cublasSetVector(N, sizeof(float), h_A ,1 ,d_A ,1);   // copy h_A to d_A
    cublasSetVector(N, sizeof(float), h_B ,1 ,d_B ,1);

    cublasSaxpy(handle, N, &alpha, d_A, 1, d_B, 1);    // B <- alpha*A + B


    cublasGetVector(N, sizeof(float), d_B, 1, h_C, 1);  // copy d_B to h_C
  ...
    cublasDestroy(handle);      // destroy cublas context

  }
//  twqcd80:/home/cuda_lecture_2021/Saxpy_1GPU/Saxpy_1gpu_cublas.cu
```

Unified Memory in CUDA

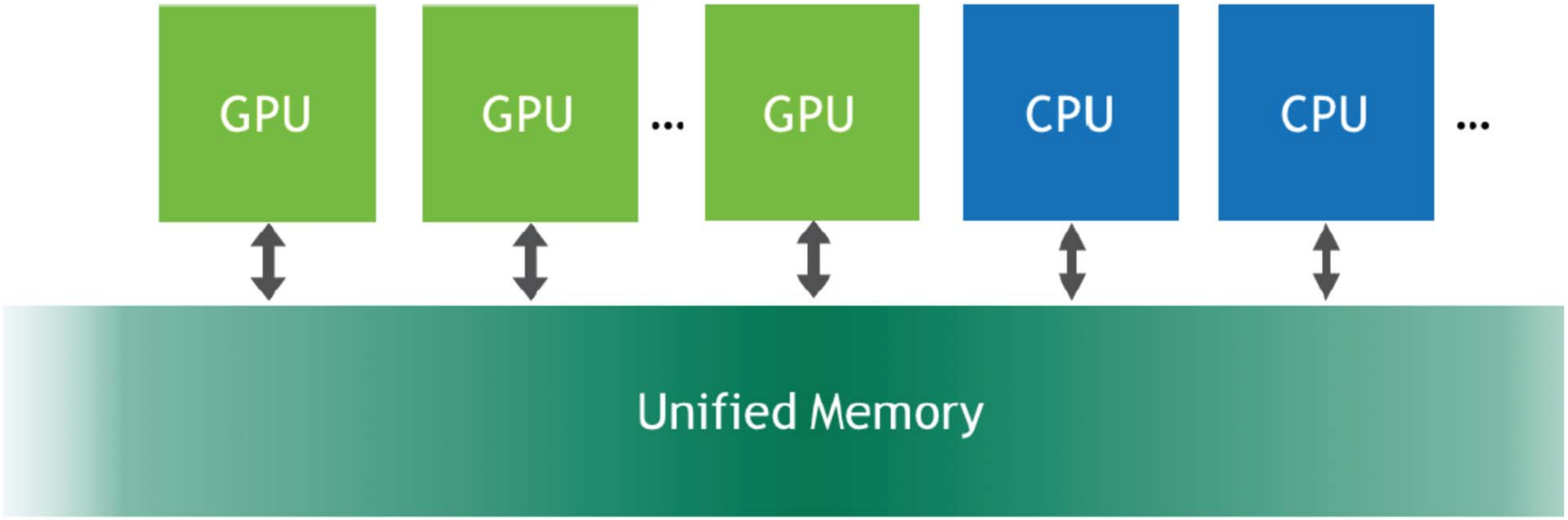


Unified Memory in CUDA is a single address space accessible from any processor in a system. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs.

```
malloc( )
cudaMalloc( )
```

```
cudaMallocManaged( )
```

Unified Memory in CUDA



Unified Memory in CUDA is a single address space accessible from any processor in a system. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs.

```
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice)
cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost)
```

SAXPY, cuBLAS with 1-GPU, unified memory

```
#include "cublas_v2.h"          // header for CUBLAS

int main(void){
...
  cublasHandle_t handle;         // CUBLAS context
...
  cudaMallocManaged(&h_A, size); // Allocate h_A and h_B in unified memory
  cudaMallocManaged(&h_B, size);
...
  cublasCreate(&handle);       // initialize CUBLAS context
  cublasSaxpy(handle, N, &alpha, h_A, 1, h_B, 1);    // B <- alpha*A + B
...
  cublasDestroy(handle);      // destroy cublas context
}

// twqcd80:/home/cuda_lecture_2021/Saxpy_1GPU/Saxpy_1gpu_cublas_umem.cu
```

Sgemm [$C \leftarrow \alpha A \cdot B + \beta C$], multi-GPUs, unified memory

```
__global__ void sgemm_umem(int n, float alpha, const float *A, const float *B,
           const float beta, float *C, const int NGPU, const int cpu_thread_id)
{
  int offset = n/NGPU*cpu_thread_id;
  int i = blockDim.x * blockIdx.x + threadIdx.x + offset;
  for(int j = 0; j < n; ++j) {
    float prod = 0.0f;
    for(int k = 0; k < n; ++k) prod += A[i*n+k]*B[k*n+j];    //  A(i,k)*B(k,j)
    C[i*n + j] = alpha*prod + beta*C[i*n + j];
  }
  __syncthreads();
}

int main(void)
{ ...
    cudaMallocManaged((void**)&h_A, size);  // allocate unified memory
    cudaMallocManaged((void**)&h_B, size);
    cudaMallocManaged((void**)&h_C, size);
  ...
  #pragma omp parallel private(cpu_thread_id){
    cpu_thread_id = omp_get_thread_num();
    cudaSetDevice(Dev[cpu_thread_id]);
    sgemm_umem<<<blocks,threads>>>(N, alpha, h_A, h_B, beta, h_C, NGPU, cpu_thread_id);
    cudaDeviceSynchronize();
  }
  ...
} // The complete code at twqcd80:/home/cuda_lecture_2021/Sgemm_NGPU/Sgemm_umem.cu
```

Sgemm, cuBLAS with multi-GPUs, unified memory

```
#include "cublasXt.h"              // header for cublasXT

int main(void){
...
  cublasXtHandle_t handle;        // cublasXt context
...
  cudaMallocManaged(&h_A, size);  // Allocate h_A, h_B, h_c in unified memory
  cudaMallocManaged(&h_B, size);
  cudaMallocManaged(&h_C, size);
...
  cublasXtCreate(&handle);        // initialize cublasXt context
  cublasXtDeviceSelect(handle, NGPU, Dev); // Select devices for CUBLASXT
  cublasXtSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, h_A, N, h_B, N,
                &beta, h_C, N);   // C <- alpha*A.B + beta*C
...
  cublasXtDestroy(handle);        // destroy cublasXt context
}

//  twqcd80:/home/cuda_lecture_2021/Sgemm_NGPU/Sgemm_ngpu_cublas.cu
```