Operating System (CSC 3150)

Tutorial 6

SHIHAO HONG › ZEYIN ZHANG

SCHOOL OF SCIENCE AND ENGINEERING

E-MAIL: 220019037@LINK.CUHK.EDU.CN

27/28, OCT 2021

Before we start -- Where to code

1. if your labtop or desktop PC is equipped with NVIDIA GPU card, you can complete your assignment on your computer.

2. Otherwise, you need to go to TC301. But be aware that there are only 40 computers avalaible there, get started as soon as possible in case that TC301 get crowded as time approaches the deadline.

3. For the international students who are not in compus and don't have a computer with NVIDIA GPU card, we are still talking about how to handle this case, we may provide a server, or allowed you to remotely get access to the computer in TC405. Anyway, it’s not determined yet, we will post notification when we make a final decision.

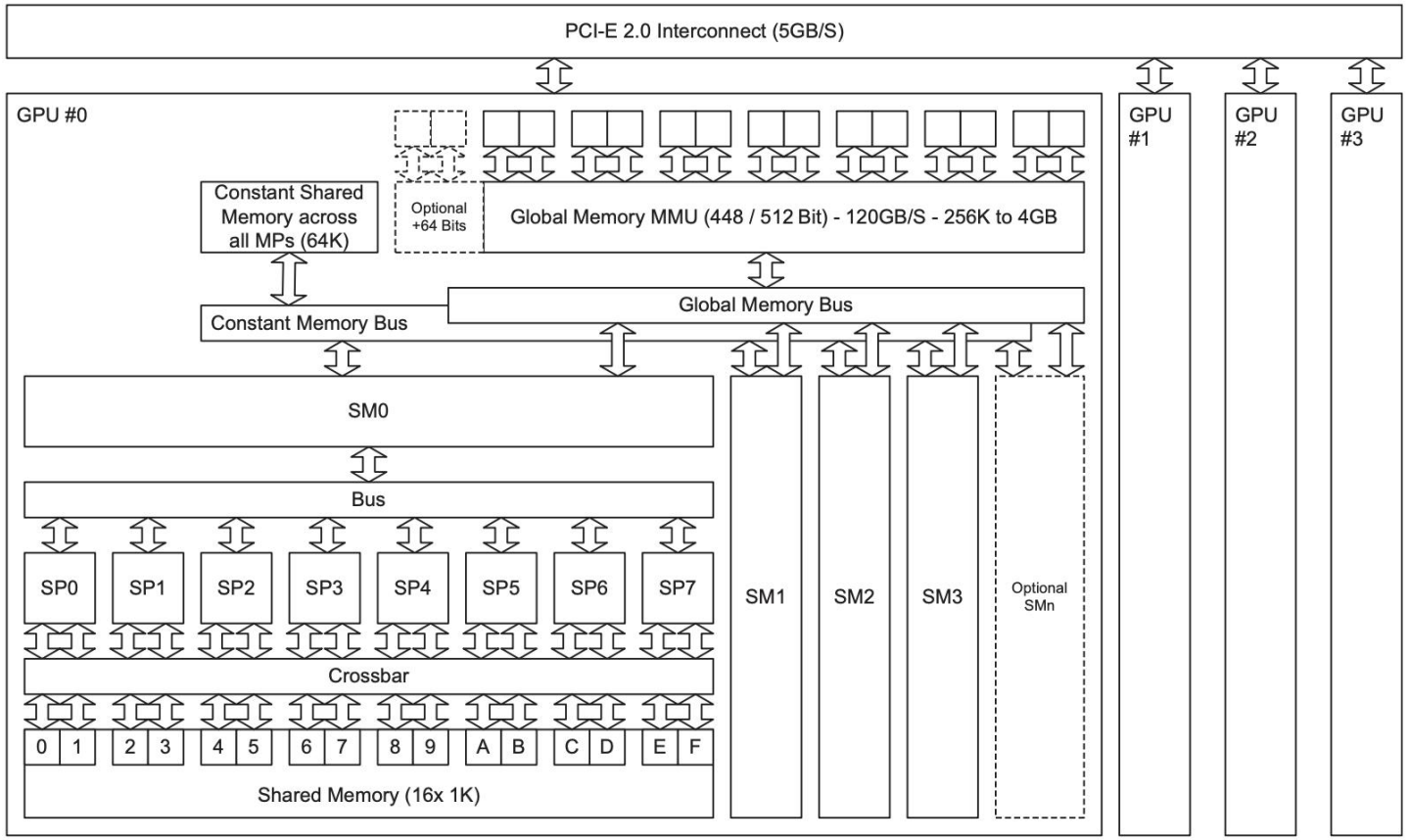
Target

In this tutorial, we will have a basic understand of Nvidia GPU Architecture, CUDA programming and paging memory management.

- GPU Memory Hierarchy and Thread Hierarchy
- CUDA programming (with C extension)
- Page Tables
- Replacement Algorithm (LRU)

Nvidia GPU Architecture

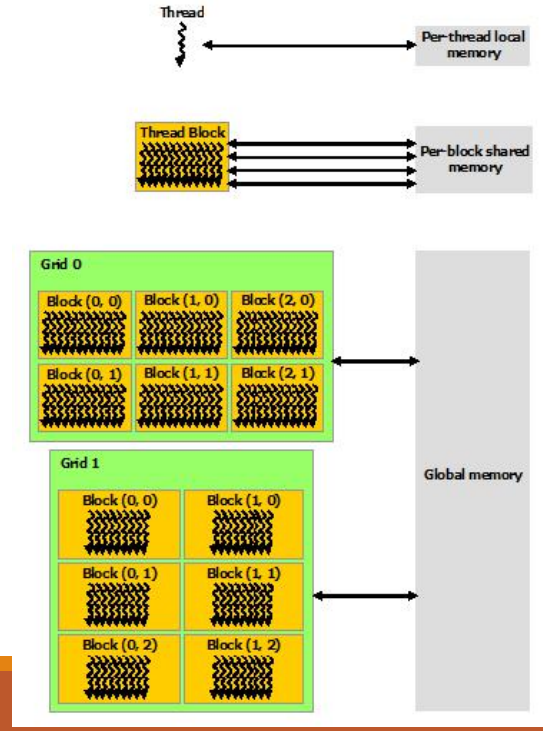
- GPU Memory
  - Global Memory
  - Shared Memory
  - etc.
- GPU Thread
  - Stream Multiprocessor
  - Stream Processor
  - etc.



NVIDIA G80 Graphic Card structure illustration

CUDA programming

- CUDA memory hierarchy
  - CUDA threads may access data from multiple memory spaces during their execution.
  - Each thread has private local memory.
  - Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.
  - All threads have access to the same global memory.



CUDA programming

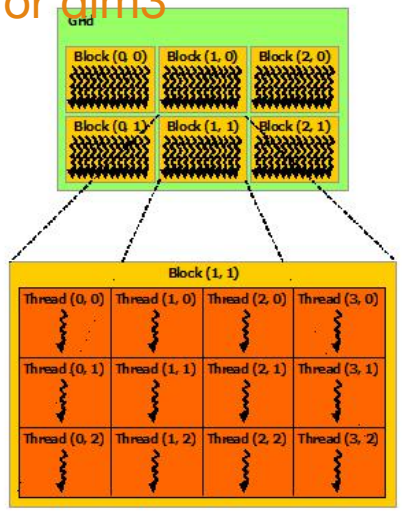
- Launching kernel
  - kernel<<<dim3 dG, dim3 dB>>>(…)

Arguments passed to kernel function.

- Execution Configuration (“<<< >>>”)
  - dG - dimension and size of grid in blocks
    Two-dimensional: x and y
    Blocks launched in the grid: dG.x * dG.y
  - dB - dimension and size of blocks in threads:
    Three-dimensional: x, y, and z
    Threads per block: dB.x * dB.y * dB.z

The number of blocks per grid and the number of threads per block specified in the <<<...>>> syntax
can be of type int or dim3

- Unspecified dim3 fields initialize to 1




```
dim3 grid, block;
grid.x=3; grid.y=2;
block.x=4; block.y=3;
kernel<<<grid, block>>>(…);
```

```
dim3 grid(3,2), block(4,3)
kernel<<<grid, block>>>(…);
```

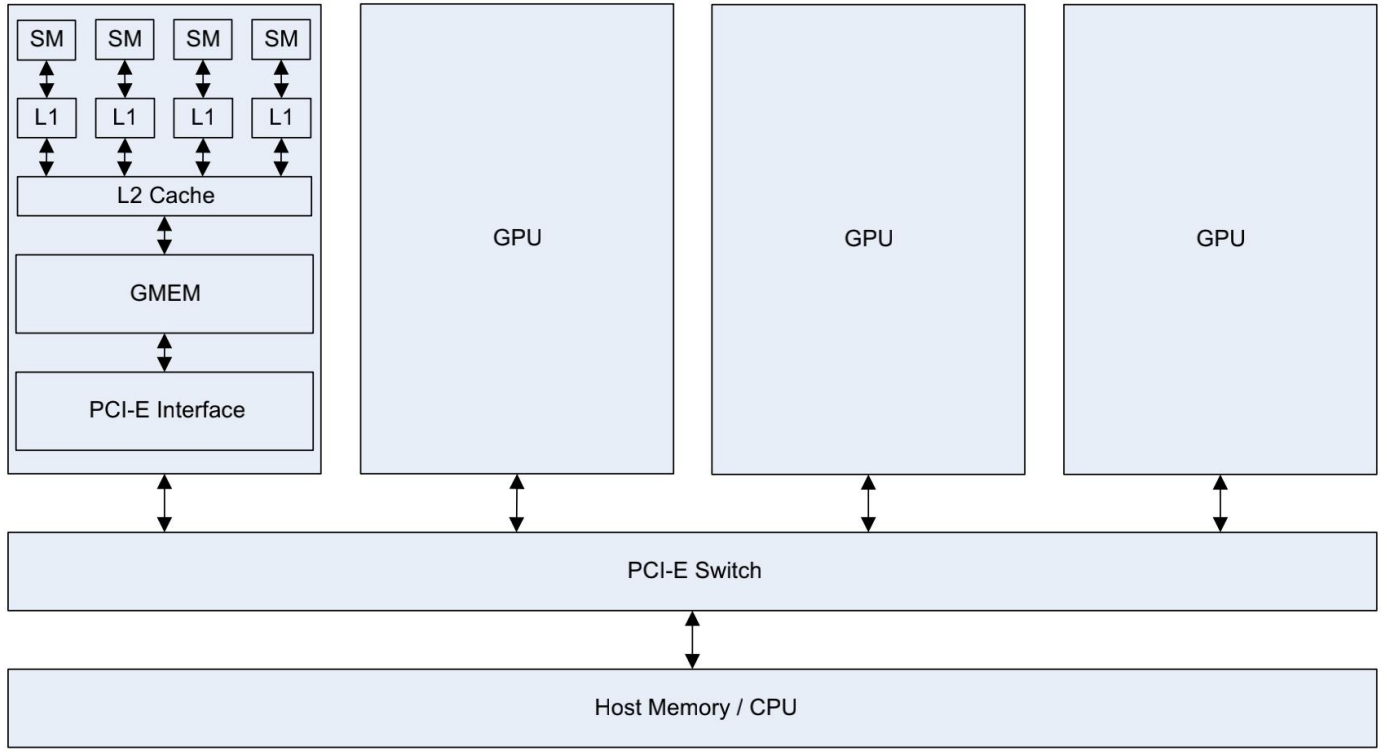
```
kernel<<<6, 12>>>(…);
```

array to parrallel them

```
__global__ void MatAdd(int A[N][N], int B[N][N], int C[N][N]){
        int i = threadIdx.x, j = threadIdx.y;
        C[i][j] = A[i][j] + B[i][j];
}

int main(){
        dim3 dimBlock(N,N);
        MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

CUDA programming

- Header should be included
  - “cuda.h”
  - “device_launch_parameters.h”
- File extension
  - “.cu”
- Definition
  - Host: CPU
  - Device: GPU
  - Kernel: function that runs on the device



CUDA programming

- Function declaration
  - __global__ CPU call GPU
    Kernels designated by function qualifier
    Function called from host and executed on device
    Must return void and cannot be a member of class.
  - __device__
    Other CUDA function qualifiers
    Functions called from device and run on device
    Cannot be called from host code
  - __host__
    Function called from host and executed on host (default)

CUDA programming

- Variable declaration
  - __device__ allocated in GPU
    Stored in global memory (large, high latency, no cache)
    Allocated with cudaMalloc (__device__ qualifier implied)
    Accessible by all threads
    Lifetime: application
  - __shared__
    Stored in on-chip shared memory (very low latency)
    Specified by execution configuration or at compile time
    Accessible by all threads in the same thread block
    Lifetime: thread block
  - __managed__ (only compute capacity greater than 3.0 could support)
    memory space specifier, optionally used together with __device__
    Can be referenced from both device and host code, e.g., its address can be taken or it can be read or written directly from a device or host function.
    Lifetime: application.
  - Unqualified variables:
    Scalars and built-in vector types are stored in registers
    What doesn't fit in registers spills to "local" memory

CUDA programming

- Choose which GPU runs on (multi-GPU system):
  - cudaSetDevice(0) (By default, set it as 0 for single-GPU system)

- Linear memory allocation in device
  - cudaMalloc()
  - cudaFree()

- Linear memory allocation in host
  - malloc()
  - free()

CUDA programming

- Data transfer between host and device memory
  - cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction)
  - direction specifies locations (host or device) of src and dst
  - enum cudaMemcpyKind:
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice
  - If the GPU compute capability is greater than 3.0, can access the variable (using __managed__ specifier) from both device and host.

CUDA Installation

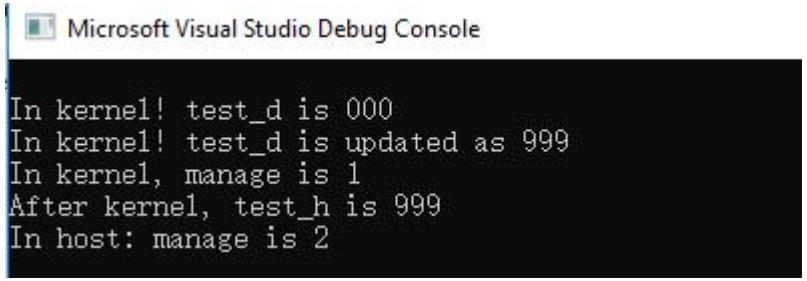
- Installation:
  - Check whether your device is available to install CUDA. (CUDA-Enabled NVIDIA GPU)
  - Check compute capability of your GPU.
    https://developer.nvidia.com/cuda-gpus
  - Check compute capability of supporting features.
    https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities
  - Select appropriate CUDA version to install.
    https://developer.nvidia.com/cuda-downloads
  - Follow the guide ‘CUDA Installation Tips.pdf’ to install CUDA.

CUDA programming

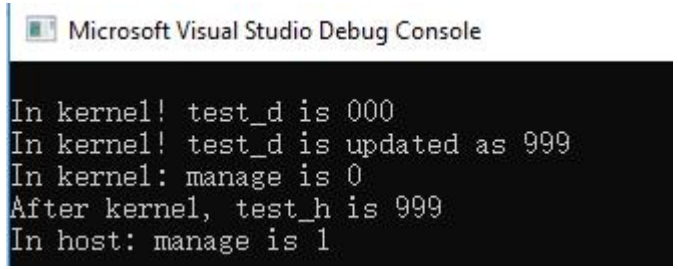
```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

#define SIZE 3
//__device__ int manage;
__device__ __managed__ int manage;

__global__ void mykernel(int *test_d)
{
    int i;

    printf("In kernel! test_d is ");
    for (i = 0; i < SIZE; i++)
    {
        printf("%d", test_d[i]);
    }
    printf("\n");

    for (i = 0; i < SIZE; i++)
    {
        test_d[i]=9;
    }

    printf("In kernel! test_d is updated as ");
    for (i = 0; i < SIZE; i++)
    {
        printf("%d", test_d[i]);
    }
    printf("\n");

    printf("In kernel: manage is %d\n", manage);
    manage = 2;
}
```

```
int main()
{
    cudaError_t cudaStatus;
    int * test_h;
    int * test_d;
    int i;
    manage = 1;

    cudaSetDevice(0);
    cudaMalloc(&test_d, sizeof(int)*SIZE);
    test_h = (int *)malloc(sizeof(int)*SIZE);

    for (i = 0; i < SIZE; i++)
    {
        test_h[i] = 0;
    }
    printf("\n");

    cudaMemcpy(test_d, test_h, sizeof(int)*SIZE, cudaMemcpyHostToDevice);

    mykernel << <1, 1 >> > (test_d);

    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "my kernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
        return 0;
    }

    cudaMemcpy(test_h, test_d, sizeof(int)*SIZE, cudaMemcpyDeviceToHost);

    printf("After kernel, test_h is ");
    for (i = 0; i < SIZE; i++)
    {
        printf("%d", test_h[i]);
    }
    printf("\n");

    cudaFree(test_d);
    free(test_h);

    cudaDeviceSynchronize();
    cudaDeviceReset();


    printf("In host: manage is %d\n", manage);

    return 0;
}
```

CUDA programming




```
//__device__ int manage;
__device__ __managed__ int manage;
```




```
__device__ int manage;
//__device__ __managed__ int manage;
```

test_d and test_h are copied via ‘cudaMemcpy()’.
manage is declared as ‘__device__ __managed__’, so that it can be accessed by both device and host.

The compile instruction is just like gcc or clang, it’s like nvcc main.cu –o main.out

Paging

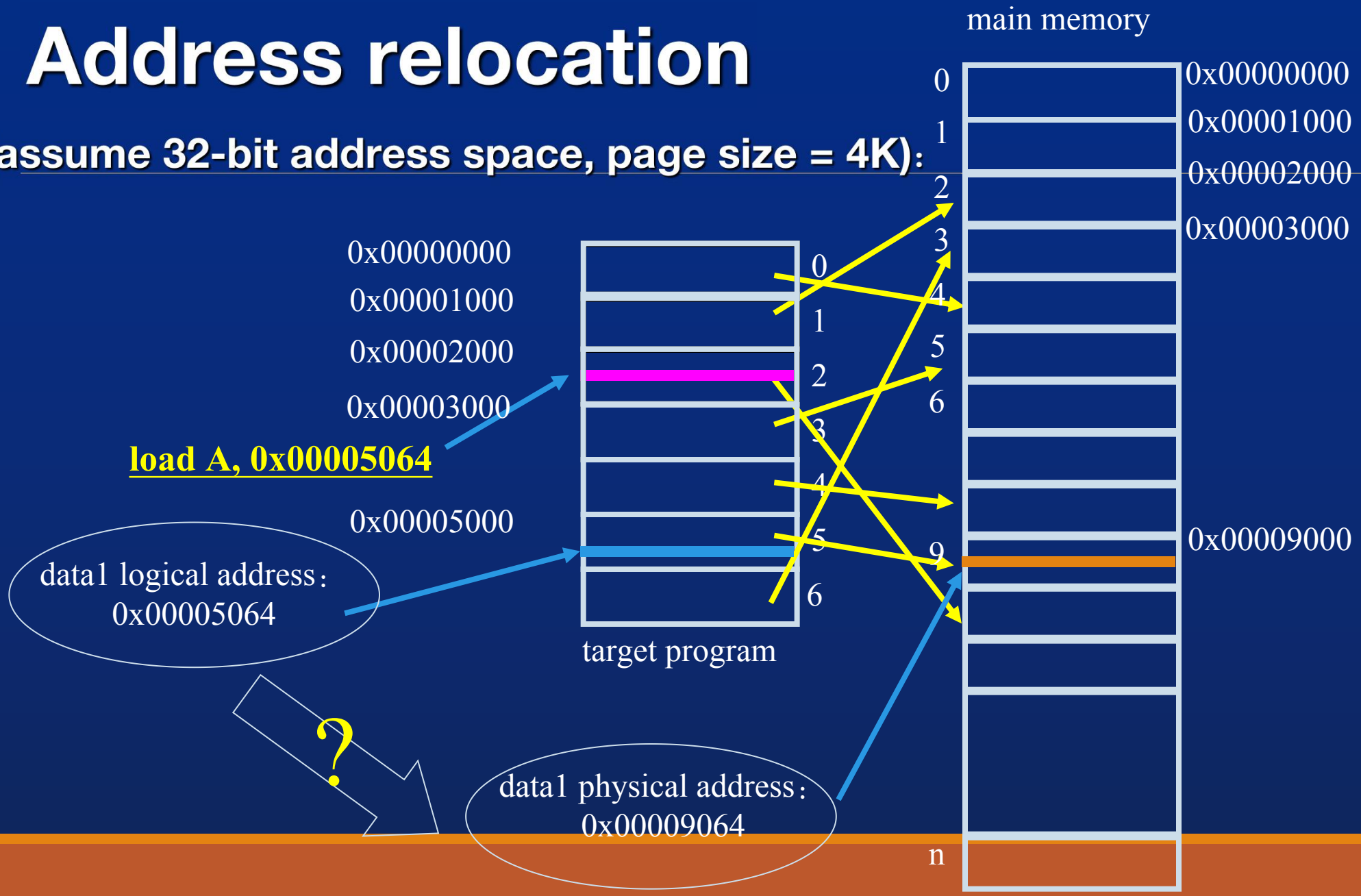
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called pages
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a page table to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

Pages Table

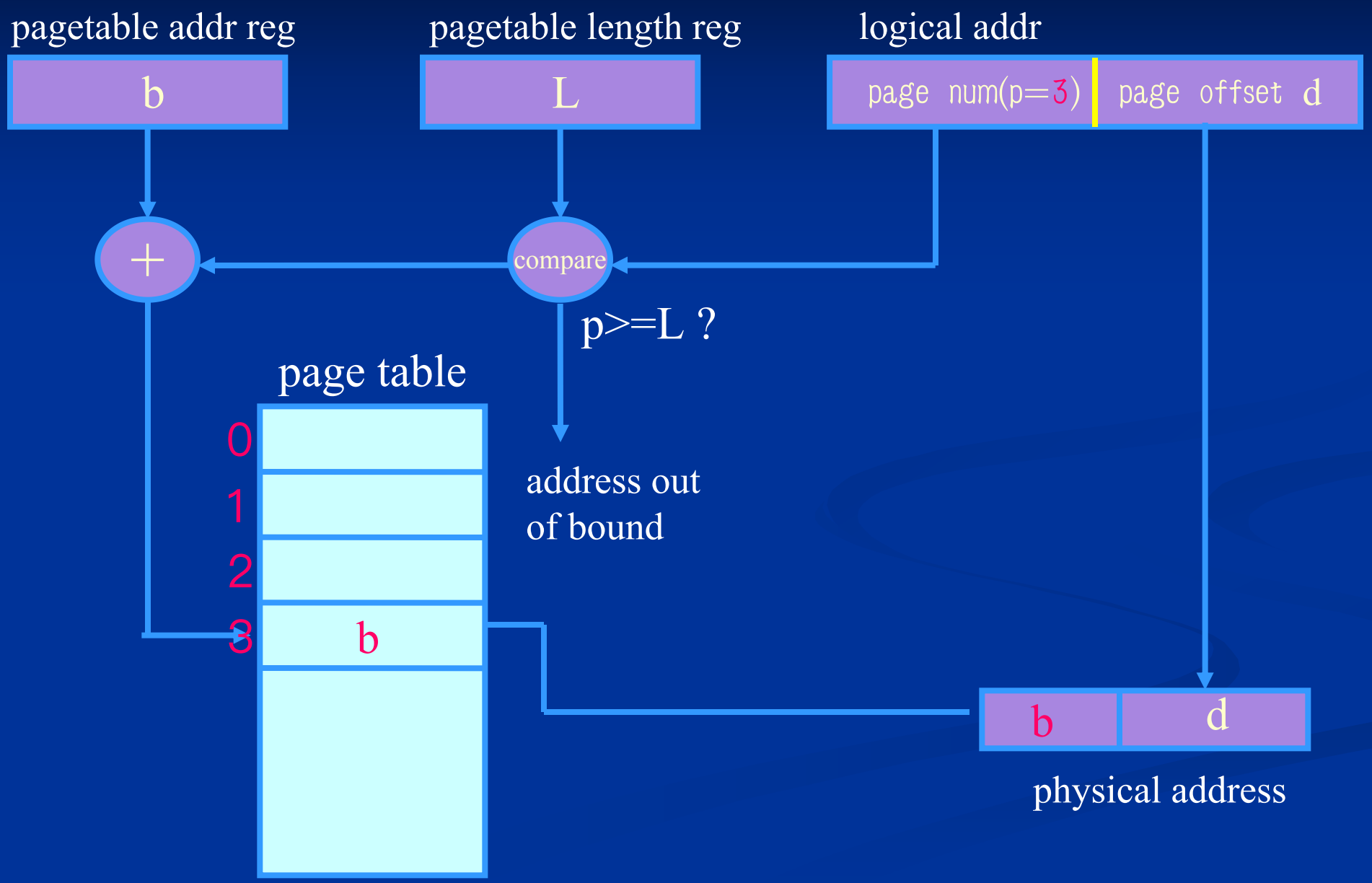
- Address generated by CPU is divided into:
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

Address relocation

(assume 32-bit address space, page size = 4K):

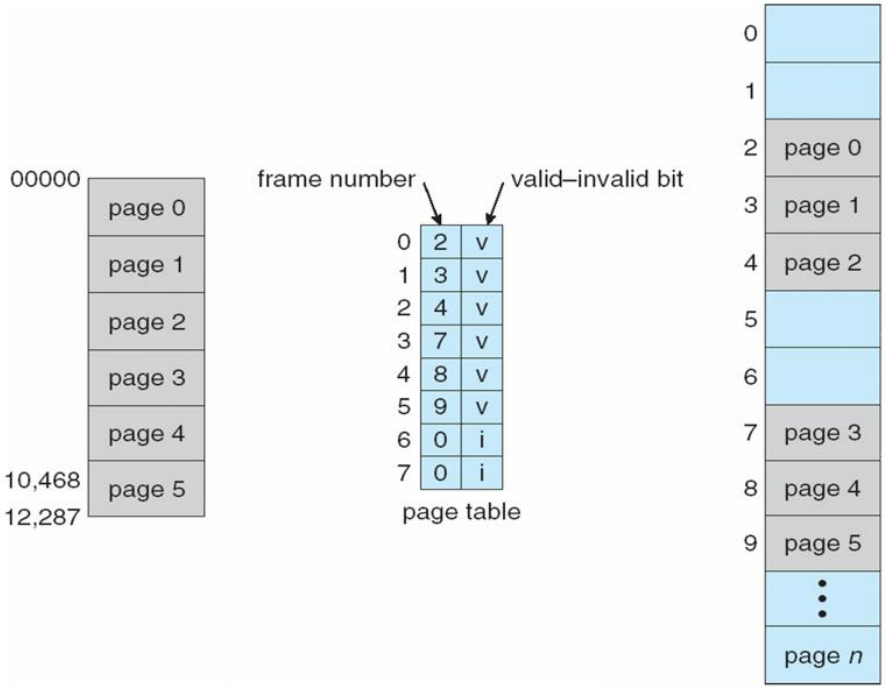


(2) Basic address mapping mechanism



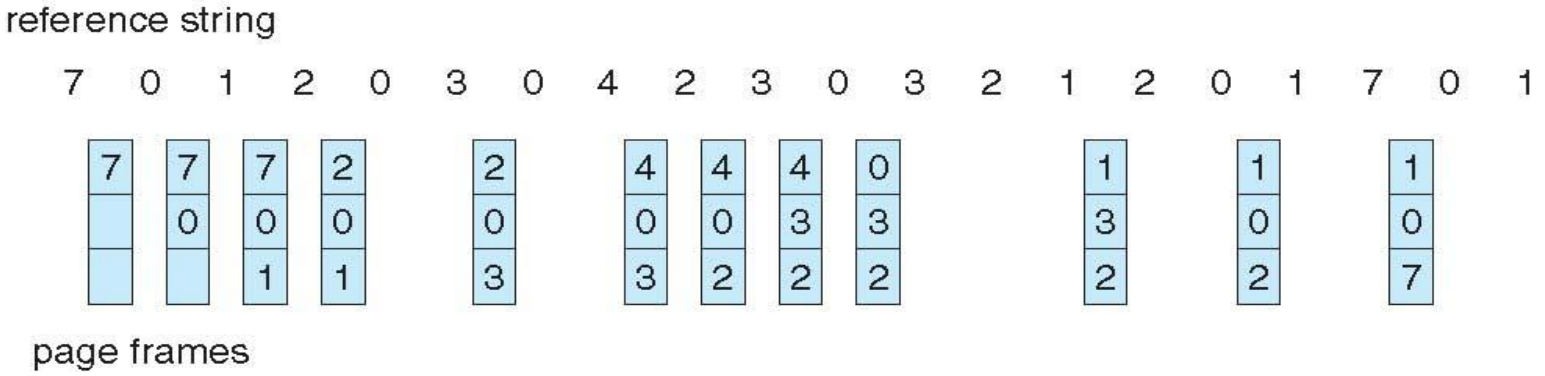
Pages Tables

- With each page table entry a valid–invalid bit is associated (v ⇒ in-memory – memory resident, i ⇒ not-in-memory)
- During address translation, if valid–invalid bit in page table entry is i ⇒ page fault



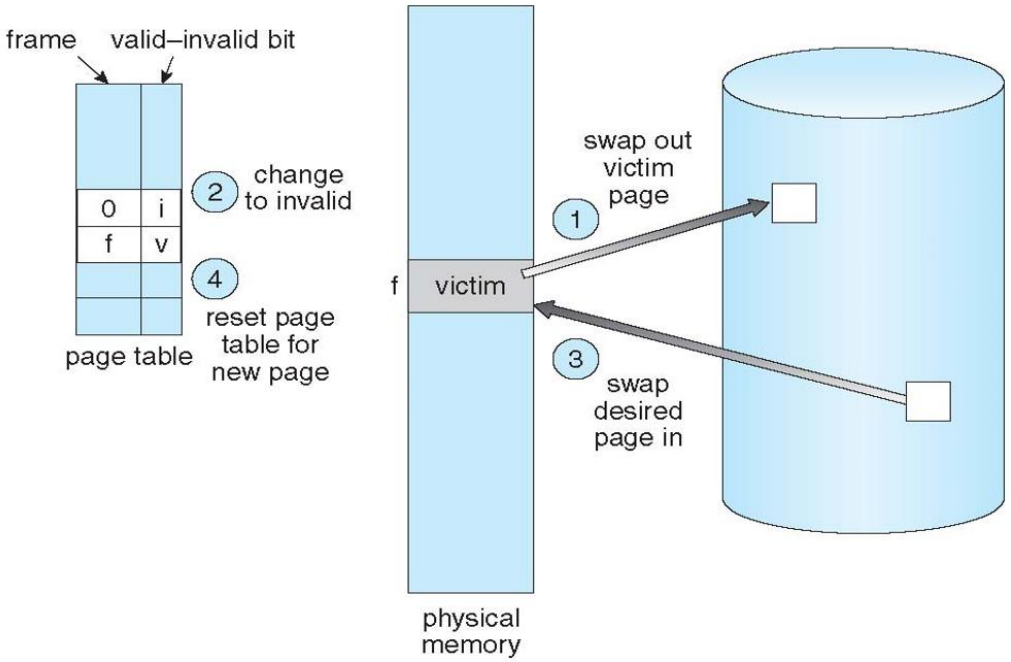
Replacement Algorithm (LRU)

- LRU: Least Recent Used
  - Use past knowledge rather than future
  - Replace page that has not been used in the most amount of time
  - Associate time of last use with each page



Page Replacement

- Page Replacement
  - When writing data to buffer, if physical memory is available, place data to available page.
  - Otherwise, replace the LRU set. Swap the least recent used page out of share memory and swap it in secondary s
  - Update the page table.



References

- CUDA toolkit document
  - https://docs.nvidia.com/cuda/archive/9.2/ (Refer to the version fits to your environment)

- CUDA programming introduction
  - https://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf
  - https://docs.nvidia.com/cuda/archive/9.2/cuda-c-programming-guide/index.html (Refer to the version of CUDA you installed.)

Thank you