GPU Teaching Kit

Accelerated Computing

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Portability and Scalability in Heterogeneous Parallel Computing

Objectives

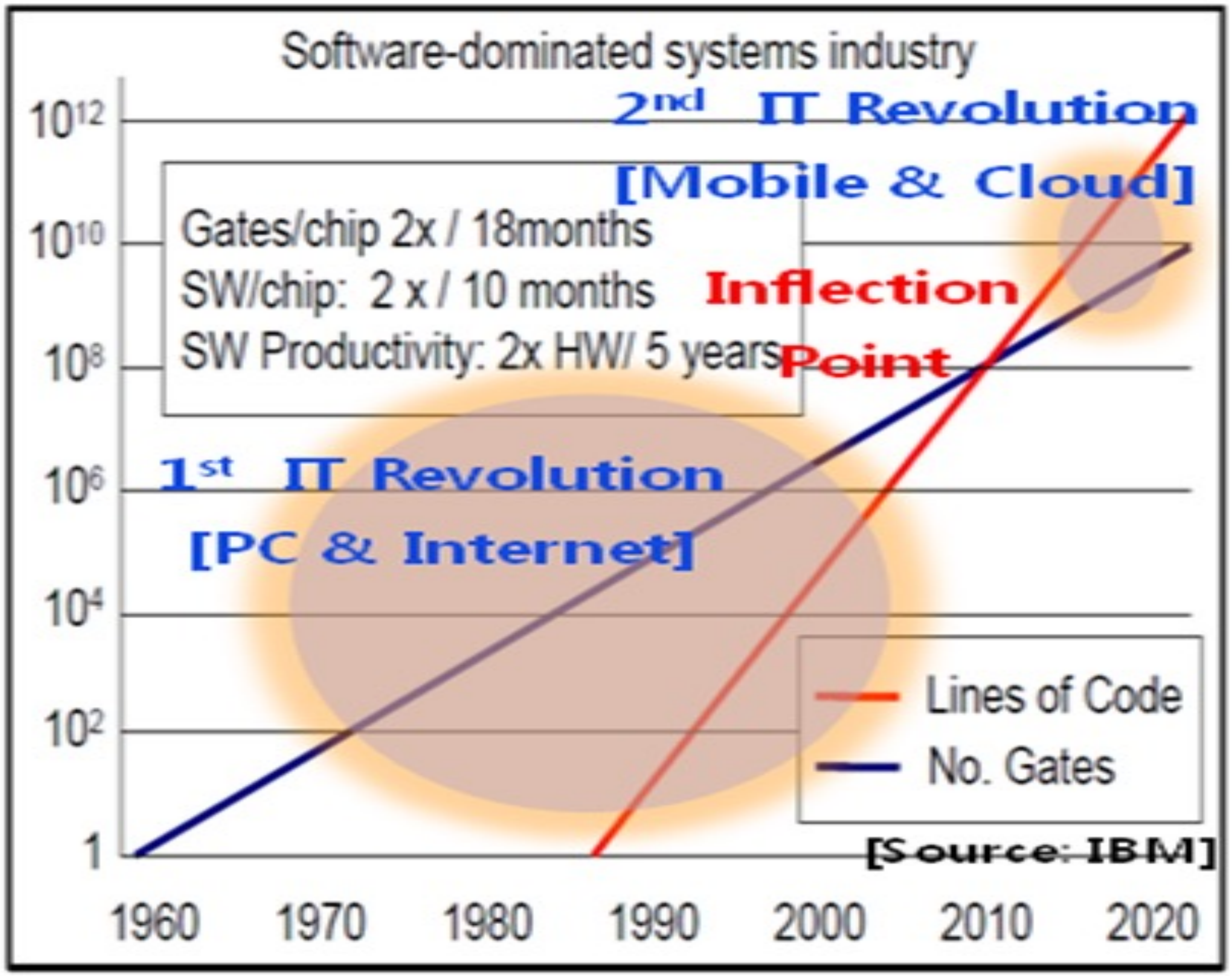
– To understand the importance and nature of scalability and portability in parallel programming

Software Dominates System Cost

– SW lines per chip increases at 2x/10 months

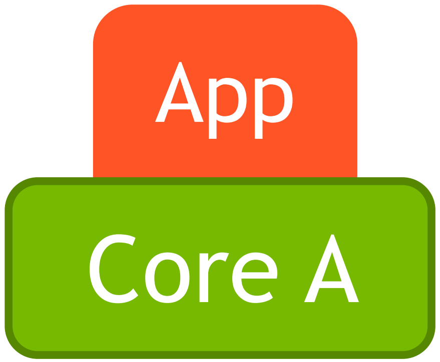
– HW gates per chip increases at 2x/18 months

– Future systems must <u>minimize software redevelopment</u>



Keys to Software Cost Control

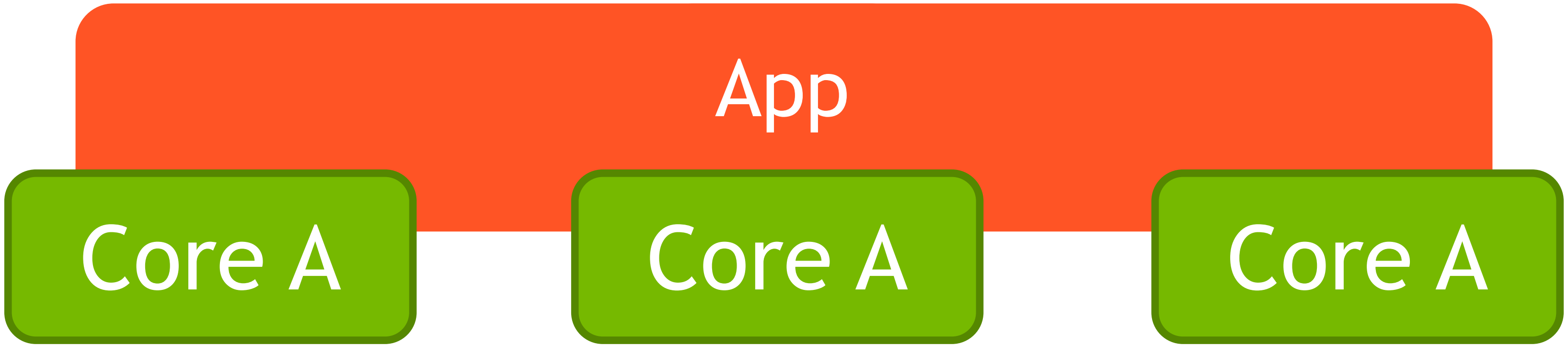
– Scalability



Keys to Software Cost Control

– Scalability
  – **The same application runs efficiently on new generations of cores**

Keys to Software Cost Control

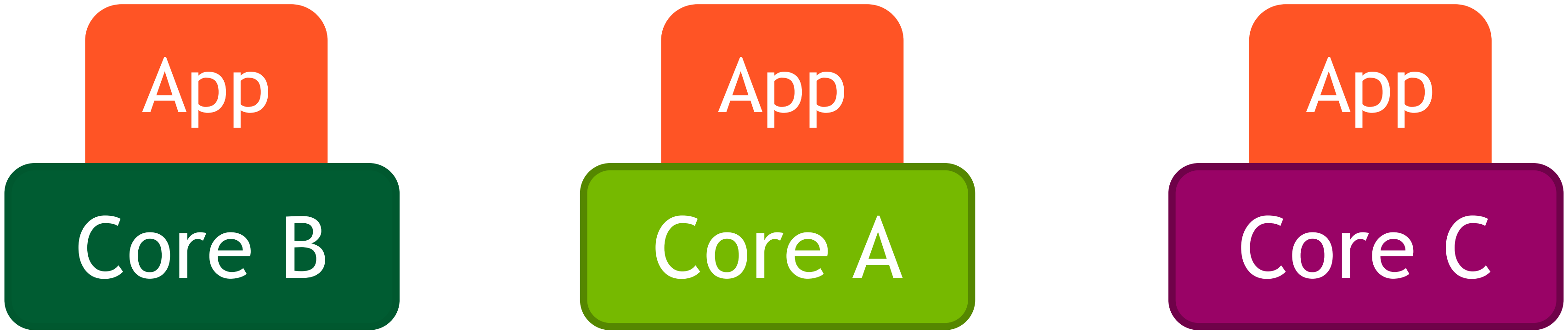


- Scalability
  - The same application runs efficiently on new generations of cores
  - **The same application runs efficiently on more of the same cores**

More on Scalability

- Performance growth with HW generations
  - Increasing number of compute units (cores)
  - Increasing number of threads
  - Increasing vector length
  - Increasing pipeline depth
  - Increasing DRAM burst size
  - Increasing number of DRAM channels
  - Increasing data movement latency

Keys to Software Cost Control



- Scalability
- **Portability**
  - The same application runs efficiently on different types of cores

Keys to Software Cost Control

- Scalability
- Portability
  - The same application runs efficiently on different types of cores
  - The same application runs efficiently on systems with different organizations and interfaces

More on Portability

- Portability across many different HW types
  - Across ISAs (Instruction Set Architectures) - X86 vs. ARM, etc.
  - Latency oriented CPUs vs. throughput oriented GPUs
  - Across parallelism models - VLIW vs. SIMD vs. threading
  - Across memory models - Shared memory vs. distributed memory

GPU Teaching Kit

Accelerated Computing

Introduction to CUDA C

CUDA C vs. Thrust vs. CUDA Libraries

Objective

- To learn the main venues and developer resources for GPU computing
  - Where CUDA C fits in the big picture

3 Ways to Accelerate Applications

Applications

Libraries	Compiler Directives	Programming Languages
Easy to use Most Performance	Easy to use Portable code	Most Performance Most Flexibility

Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

NVIDIA GPU Accelerated Libraries

DEEP LEARNING	cuDNN	TensorRT	DeepStream SDK
LINEAR ALGEBRA	cuBLAS	cuSPARSE	cuSOLVER
SIGNAL, IMAGE, VIDEO	cuFFT	NVIDIA NPP	CODEC SDK
PARALLEL ALGORITHMS	nvGRAPH	NCCL	Thrust

Vector Addition in Thrust

```
#include <thrust/device_vector.h>
#include <thrust/copy.h>

int main(void) {
  size_t inputLength = 500;
  thrust::host_vector<float> hostInput1(inputLength);
  thrust::host_vector<float> hostInput2(inputLength);
  thrust::device_vector<float> deviceInput1(inputLength);
  thrust::device_vector<float> deviceInput2(inputLength);
  thrust::device_vector<float> deviceOutput(inputLength);

  thrust::copy(hostInput1.begin(), hostInput1.end(), deviceInput1.begin());
  thrust::copy(hostInput2.begin(), hostInput2.end(), deviceInput2.begin());

  thrust::transform(deviceInput1.begin(), deviceInput1.end(),
                    deviceInput2.begin(), deviceOutput.begin(),
                    thrust::plus<float>());
}
```

Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement

- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages

- **Uncertain:** Performance of code can vary across compiler versions

OpenACC

– Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
    copyout(output[0:inputLength])
    for(i = 0; i < inputLength; ++i) {
        output[i] = input1[i] + input2[i];
    }
```

Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** Programmer has best control of parallelism and data movement

- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types

- **Verbose:** The programmer often needs to express more details

GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Python	PyCUDA, Numba
Fortran	CUDA Fortran, OpenACC
C	CUDA C, OpenACC
C++	CUDA C++, Thrust
C#	Hybridizer

CUDA - C

Applications

Libraries

Compiler Directives

Programming Languages

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

GPU Teaching Kit

Accelerated Computing

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

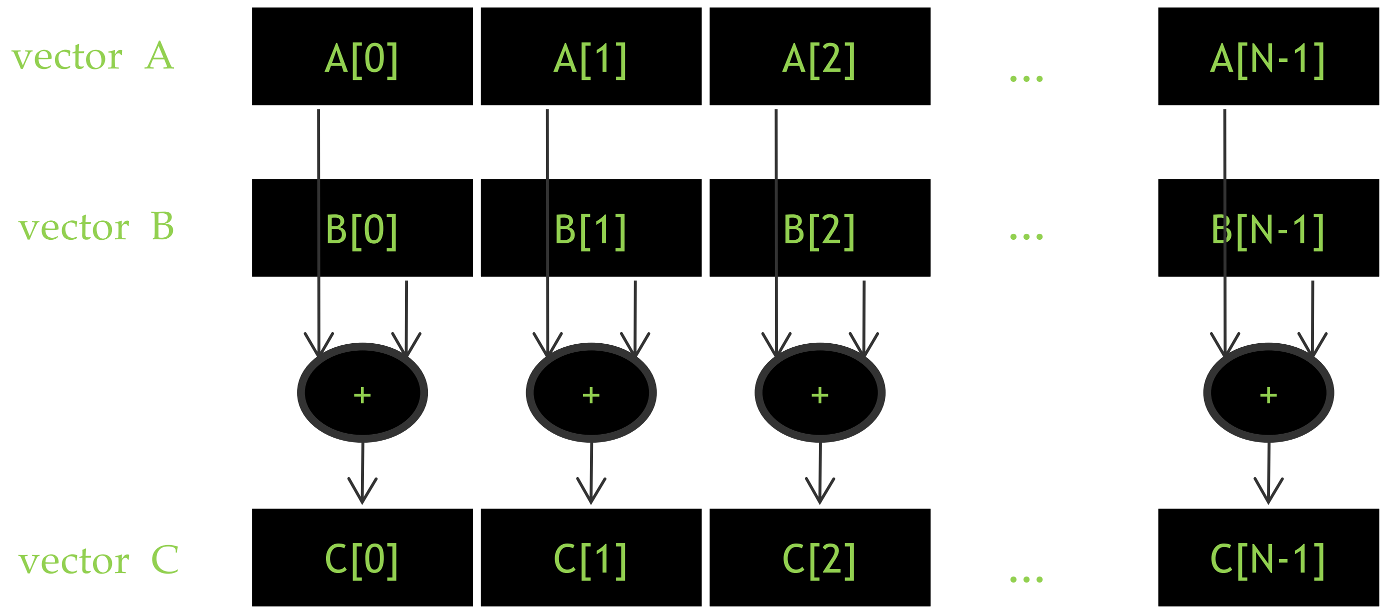
Introduction to CUDA C

Memory Allocation and Data Movement API Functions

Objective

- To learn the basic API functions in CUDA host code
  - Device Memory Allocation
  - Host-Device Data Transfer

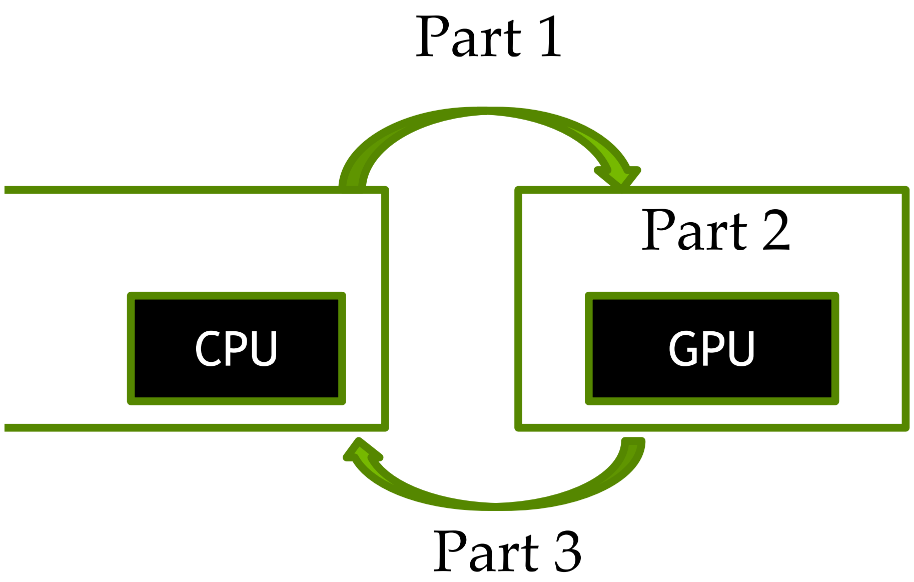
Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

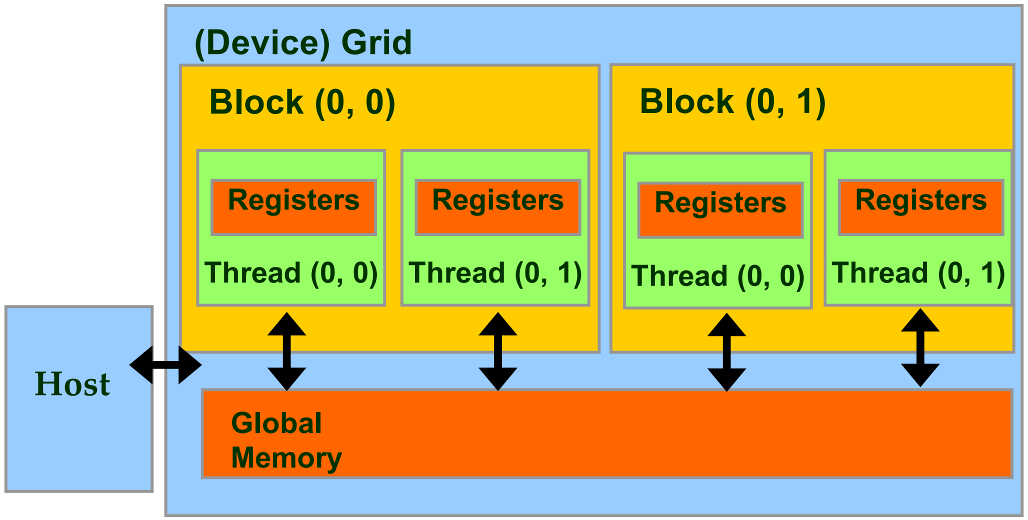
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

Heterogeneous Computing vecAdd CUDA Host Code




```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n* sizeof(float);
   float *d_A, *d_B, *d_C;
   // Part 1
   // Allocate device memory for A, B, and C
   // copy A and B to device memory

   // Part 2
   // Kernel launch code – the device performs the actual vector addition

   // Part 3
   // copy C from the device memory
   // Free device vectors
}
```

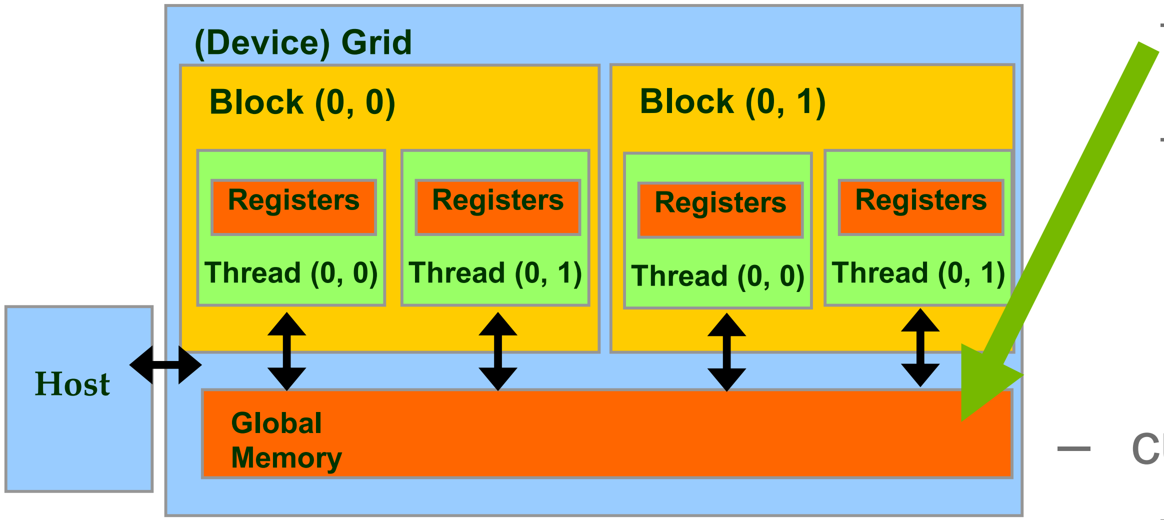
Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

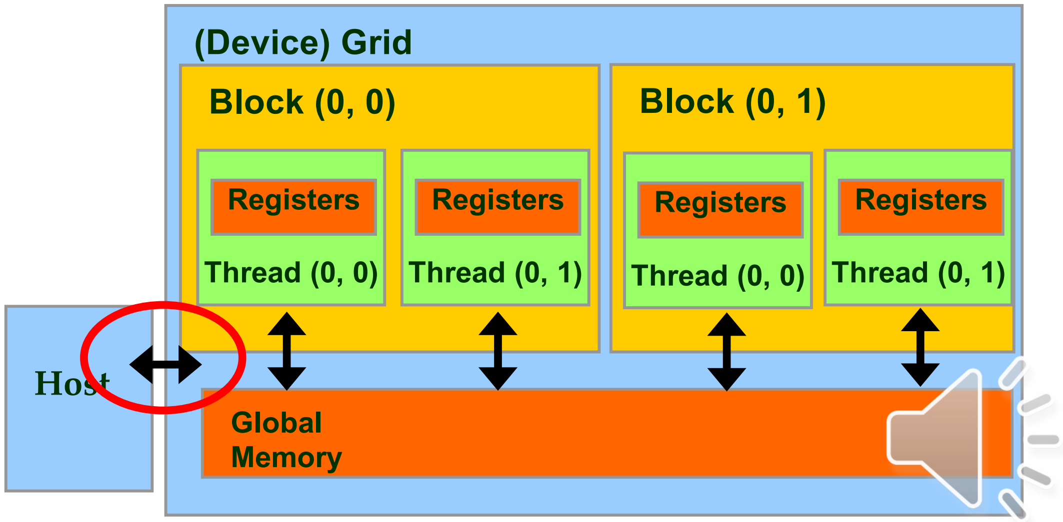
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

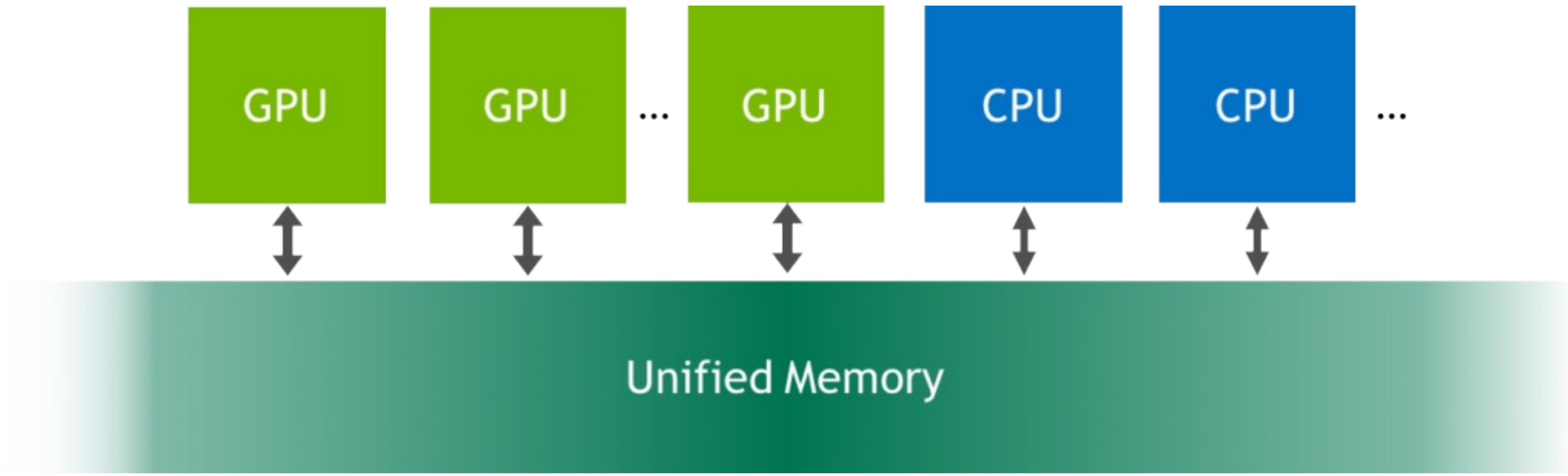
Host-Device Data Transfer API functions



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is synchronous with respect to the host

CUDA Unified Memory (UM)

- Is a single memory address space accessible both from the host and from the device.
- The hardware/software handles automatically the data migration between the host and the device maintaining consistency between them.



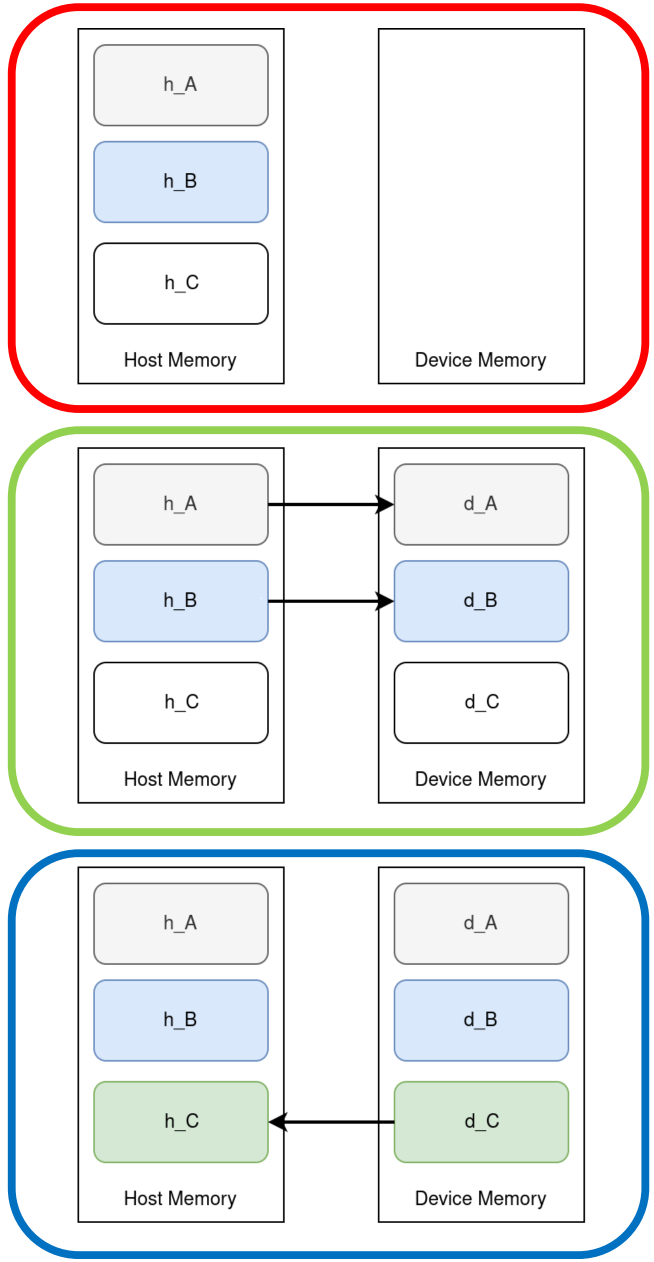
30

Vector Addition, Explicit Memory Management

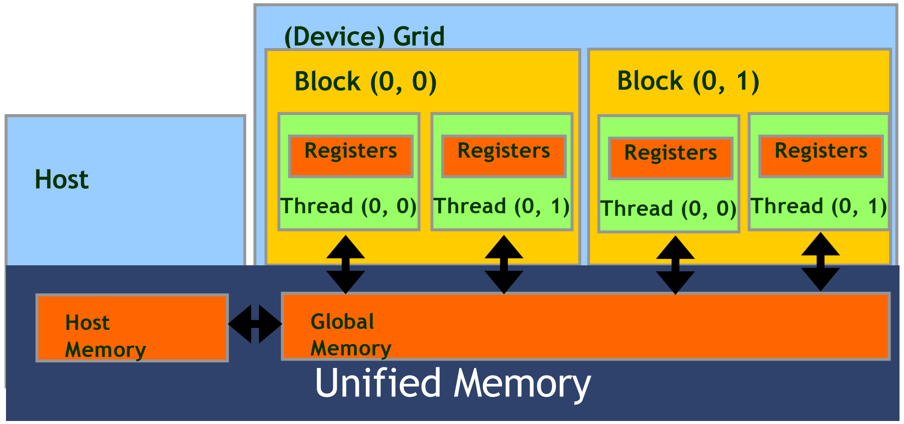
… Allocate h_A, h_B, h_C …

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n * sizeof(float); float *d_A, *d_B, *d_C;

   cudaMalloc((void **) &d_A, size);
   cudaMalloc((void **) &d_B, size);
   cudaMalloc((void **) &d_C, size);

   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
   cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

… Free h_A, h_B, h_C …

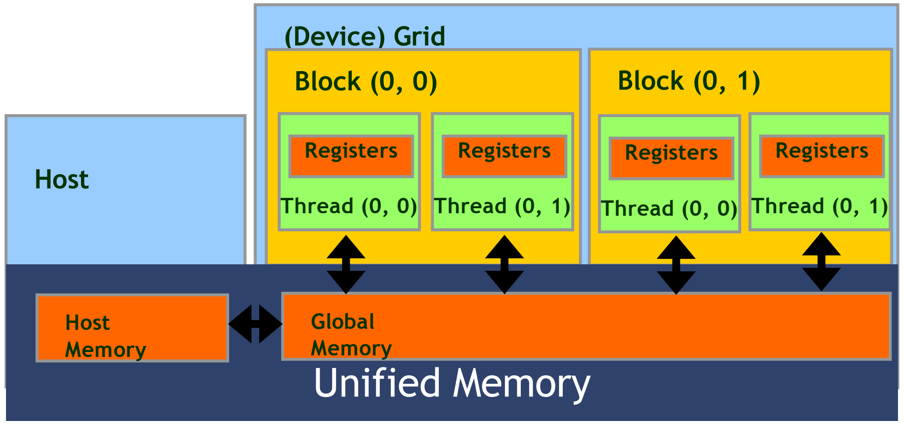


Partial Overview of CUDA Memories



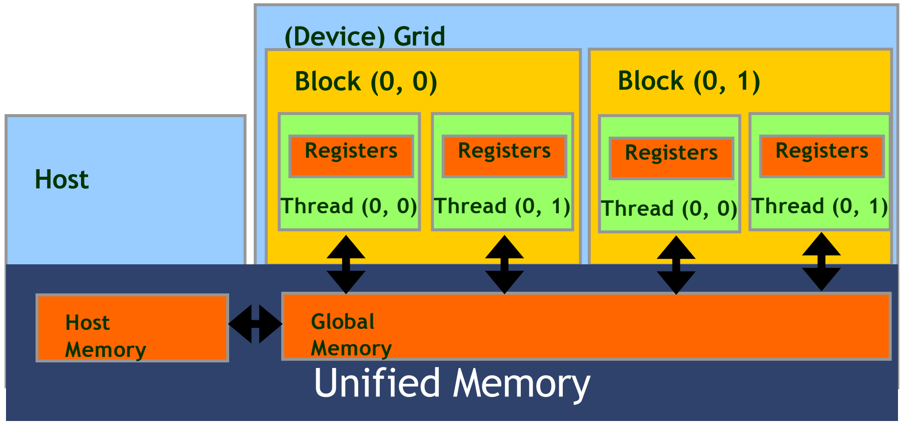
- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
  - R/W managed memory (Unified Memory)
- Host code can
  - Transfer data to/from per grid global memory
  - R/W managed memory

Partial Overview of CUDA Memories



- cudaMallocManaged()
  - Allocates an object in the Unified Memory address space.
  - Two parameters, with an optional third parameter.
    - Address of a pointer to the allocated object
    - Size of the allocated object in terms of bytes
    - [Optional] Flag indicating if memory can be accessed from any device or stream
- cudaFree()
  - Frees object from unified memory.
  - One parameter
    - Pointer to freed object

Partial Overview of CUDA Memories



- cudaMemcpy()
  - Memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Depending on the transfer type, the driver may decide to use the memory on the host or the device.
  - In Unified Memory this function is utilized to copy data between different arrays, regardless of position.

Putting it all together, vecAdd CUDA host code using Unified Memory

```
int main() {
   float *m_A, float *m_B, float *m_C, int n;

   int size = n * sizeof(float);

   cudaMallocManaged((void**) &m_A, size);
   cudaMallocManaged((void**) &m_B, size);
   cudaMallocManaged((void**) &m_C, size);

   // Memory initialization on the Host

   // Kernel invocation code - to be shown later

   cudaFree(m_A); cudaFree(m_B); cudaFree(m_C);
}
```

Allocation of Managed Memory

m_A, m_B gets initialized on the host

The device performs the actual vector addition

CUDA Unified Memory for different architectures

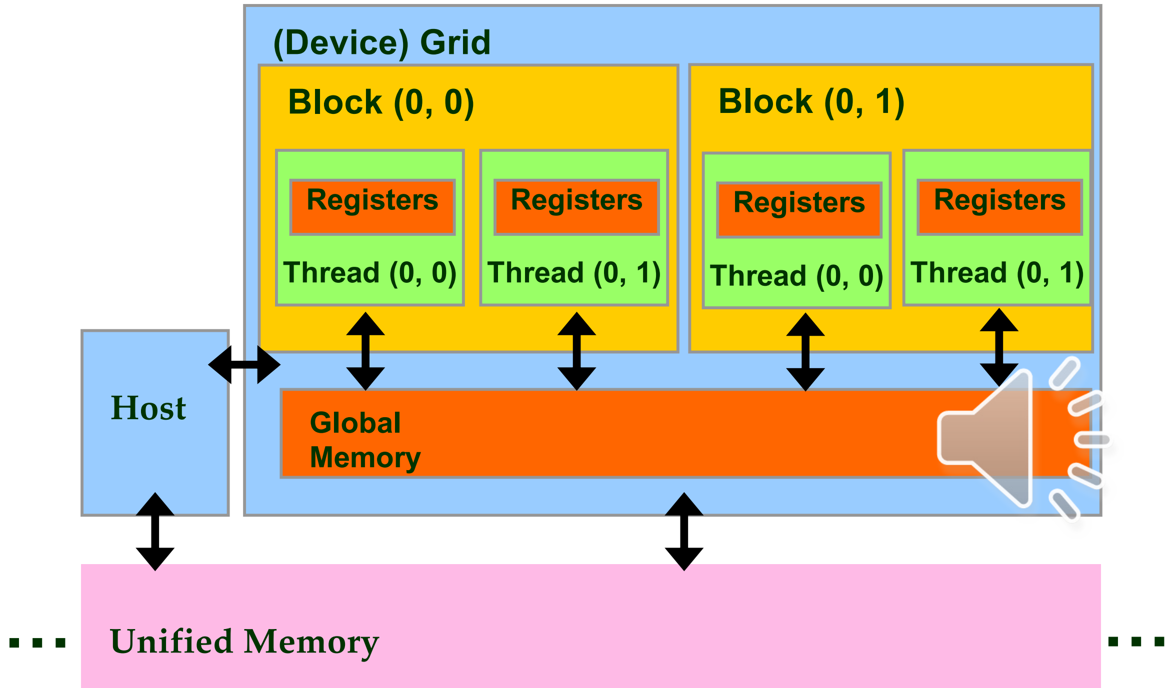
Prior to compute capability 6.x

- There is no specialized hardware units to improve UM efficiency.
- For data migration the full memory block needs to be copied synchronically by the driver.
- No memory oversubscription.

Compute capability 6.x onwards

- There are specialized hardware units managing page faulting.
- Data is migrated on demand, meaning that data gets copied only on page fault.
- Possibility to oversubscribe memory, enabling larger arrays than the device memory size.

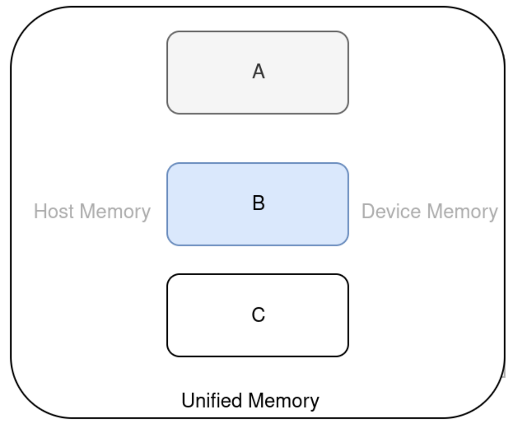
Unified Memory



- cudaMallocManaged(void** ptr, size_t size)
  - Single memory space for all CPUs/GPUs
    - Maintain single copy of data
  - CUDA-managed data
    - On-demand page migration
  - Compatible with cudaMalloc(), cudaFree()
  - Can be optimized
    - cudaMemAdvise(), cudaMemPrefetchAsync(),
    - cudaMemcpyAsync()

Vector Addition, Unified Memory

```
float *A, *B, *C
cudaMallocManaged(&A, n * sizeof(float));
cudaMallocManaged(&B, n * sizeof(float));
cudaMallocManaged(&C, n * sizeof(float));

// Initialize A, B

void vecAdd(float *A, float *B, float *C, int n)
{
 // Kernel invocation code – to be shown later
}

cudaFree(A);
cudaFree(B);
cudaFree(C);
```



In Practice, Check for API Errors in Host Code

```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

GPU Teaching Kit

Accelerated Computing