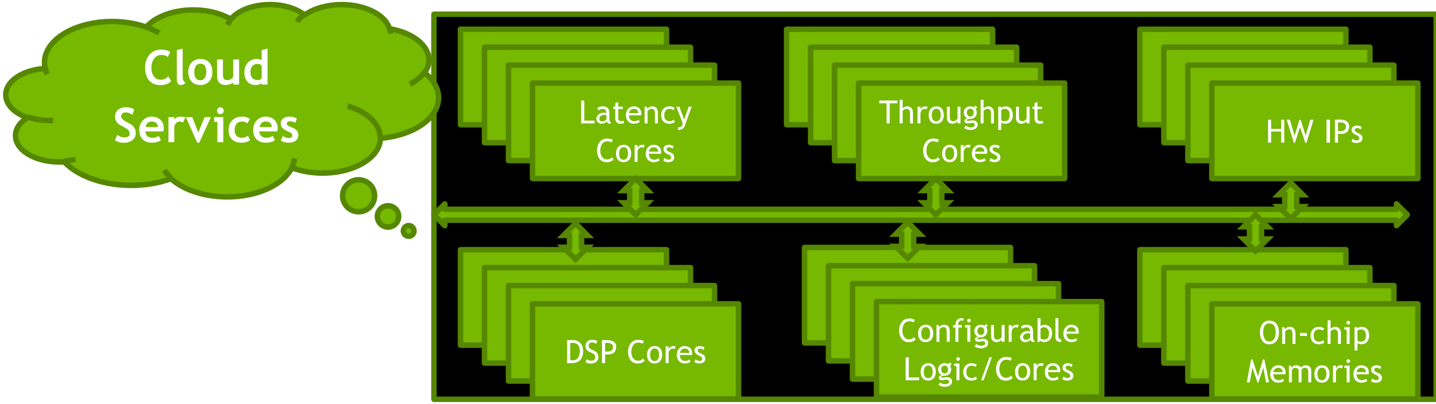
Heterogeneous Parallel Computing

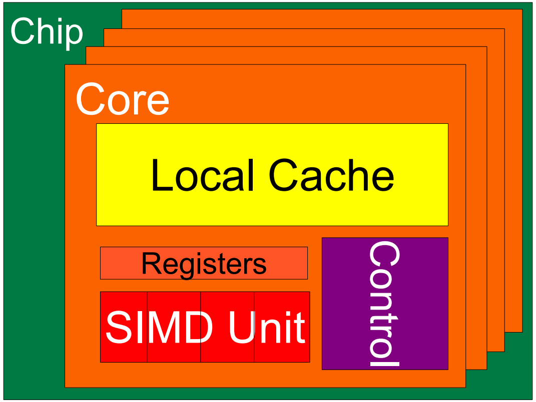
– Use the best match for the job (heterogeneity in mobile SOC)



CPU and GPU are designed very differently

CPU

Latency Oriented Cores



Chip

Core

Local Cache

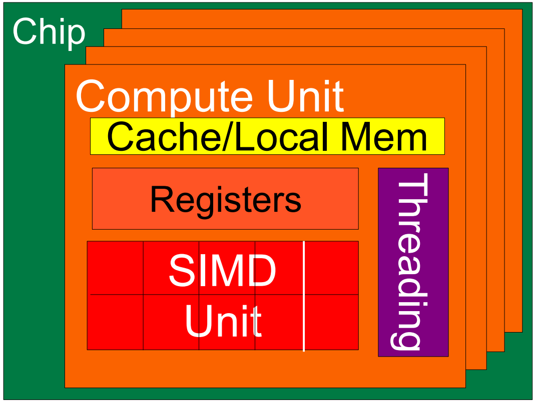
Registers

SIMD Unit

Control

GPU

Throughput Oriented Cores



Chip

Compute Unit

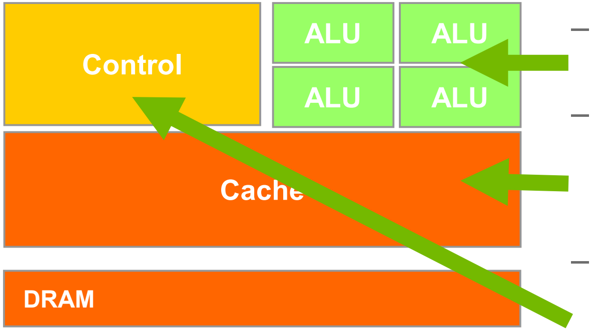
Cache/Local Mem

Registers

SIMD Unit

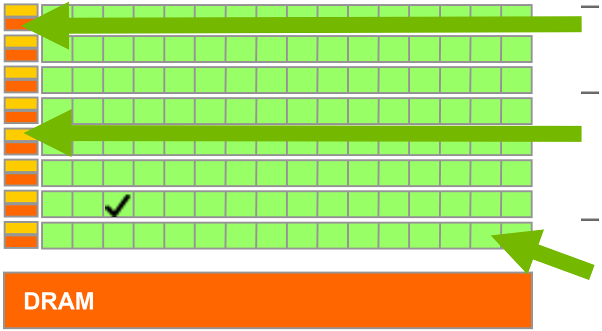
Threading

CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design



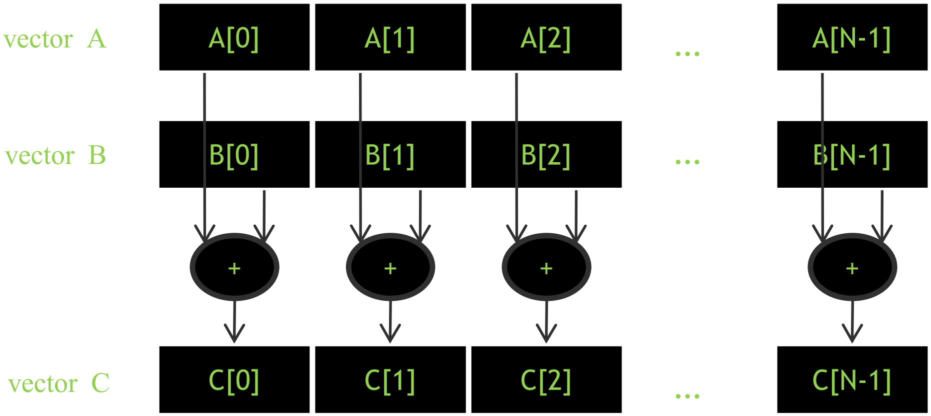
- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10X+ faster than GPUs for sequential code

- GPUs for parallel parts where throughput wins
  - GPUs can be 10X+ faster than CPUs for parallel code

Data Parallelism - Vector Addition Example



vector A A[0] A[1] A[2] … A[N-1]

vector B B[0] B[1] B[2] … B[N-1]

\+ \+ \+ \+

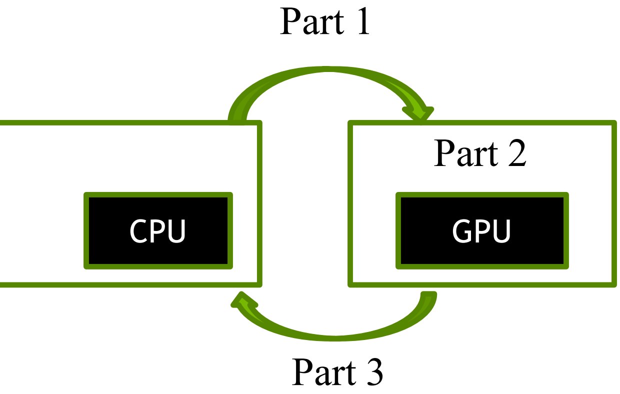
vector C C[0] C[1] C[2] … C[N-1]

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


Part 1

Part 2

CPU

GPU

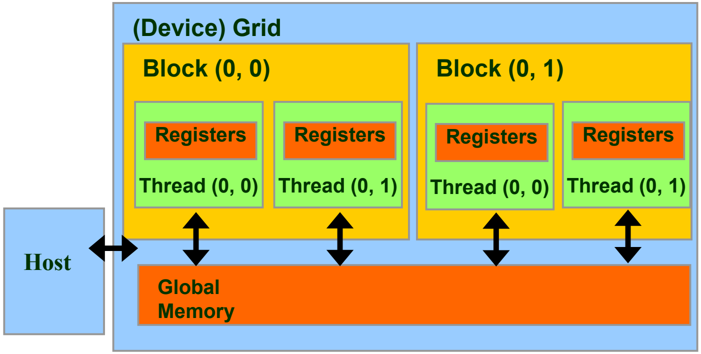
Part 3

```
#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
}
```

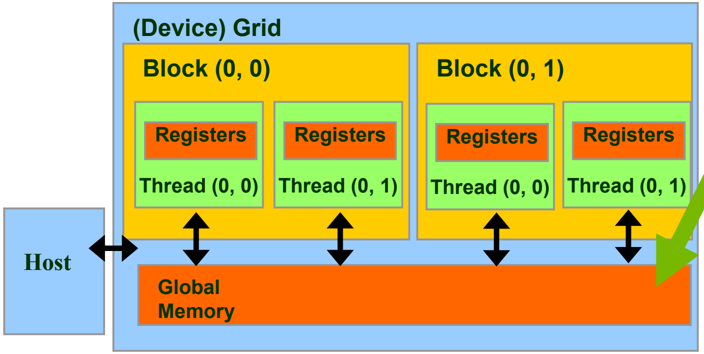
Partial Overview of CUDA Memories



- 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

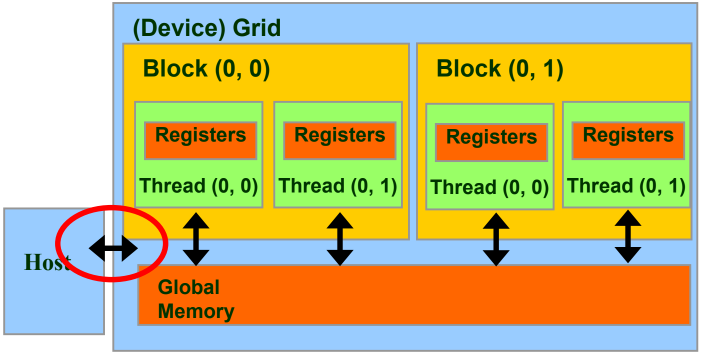
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointer** 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 asynchronous

Vector Addition Host Code

```
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);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code – to be shown later


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

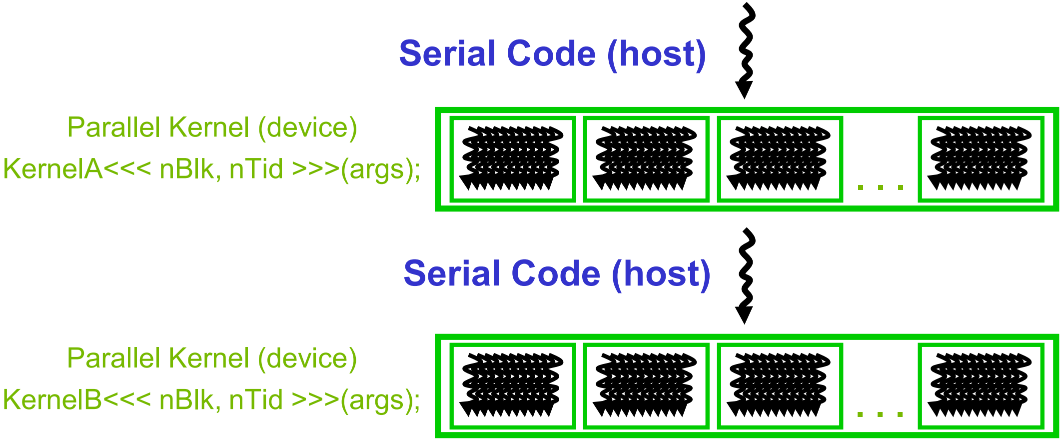
In Practice, Check for API Errors in Host Code

```
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);
}
```

CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
  - Serial parts in **host** C code
  - Parallel parts in **device** SPMD kernel code



Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

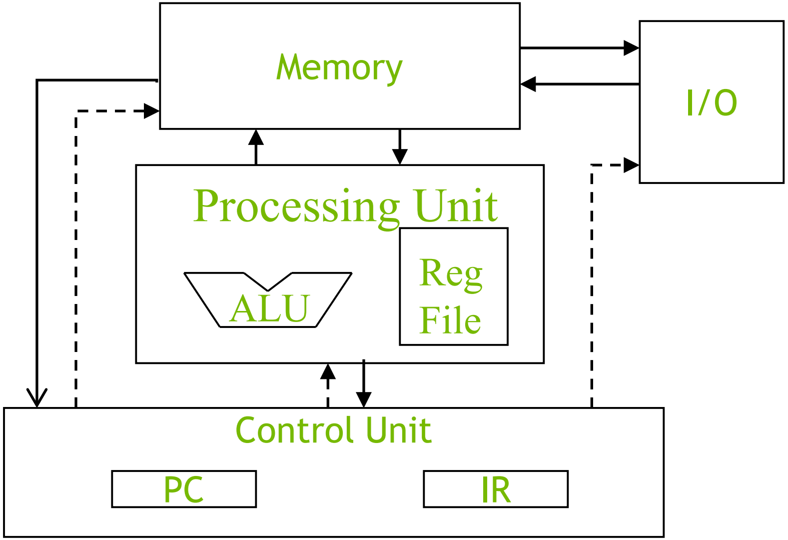
A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
  - Both CPUs and GPUs are designed based on (different) instruction sets

- Program instructions operate on data stored in memory and/or registers.

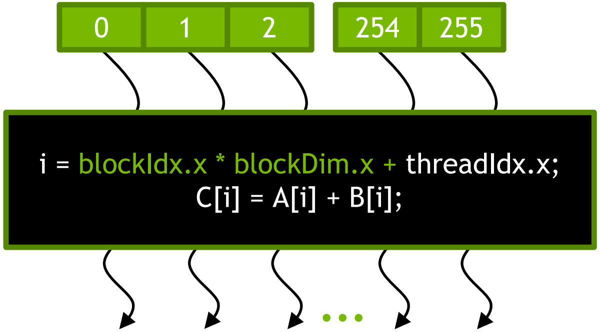
A Thread as a Von-Neumann Processor

A thread is a “virtualized” or
“abstracted”
Von-Neumann Processor



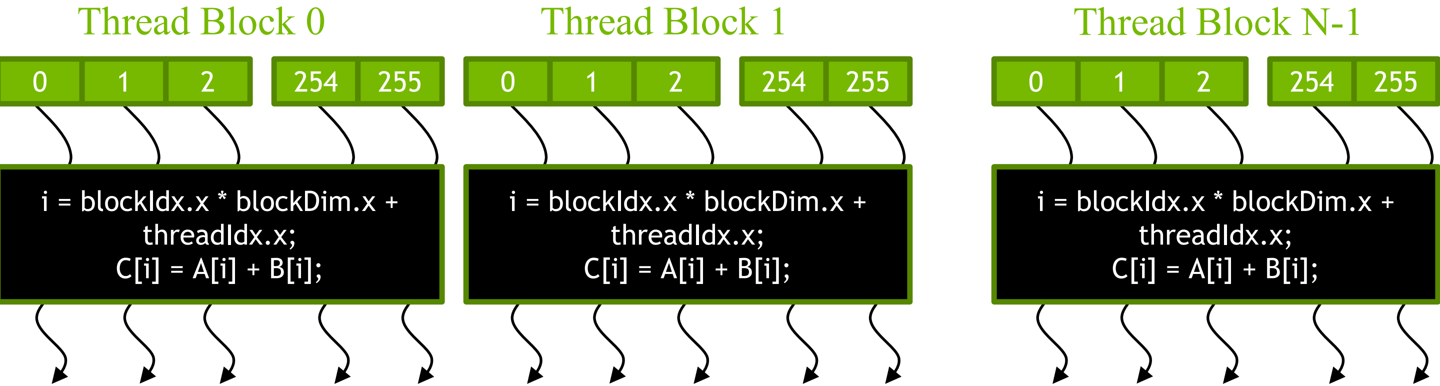
Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

Thread Blocks: Scalable Cooperation



Thread Block 0

```
i = blockIdx.x * blockDim.x +
threadIdx.x;
C[i] = A[i] + B[i];
```

Thread Block 1

```
i = blockIdx.x * blockDim.x +
threadIdx.x;
C[i] = A[i] + B[i];
```

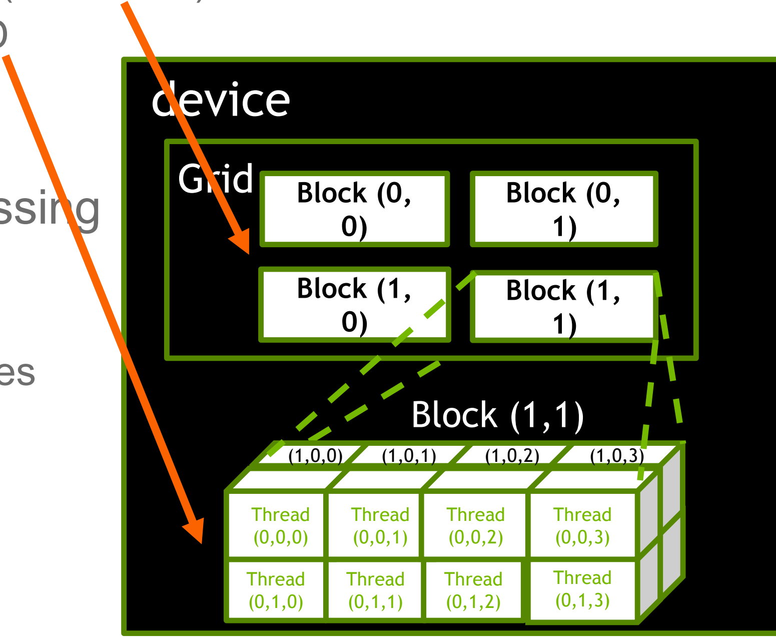
Thread Block N-1

```
i = blockIdx.x * blockDim.x +
threadIdx.x;
C[i] = A[i] + B[i];
```

- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  -



GPU Teaching Kit

Accelerated Computing