INTRODUCTION TO GRAPHICS PROCESSING UNITS

DR. CARL E. FIELDS

(he/him)

RPF Fellow, CCS-2/XCP-2
Los Alamos National Laboratory

Session 15, LSSTC DSFP
CfA, Harvard
July 22, 2022

Los Alamos
NATIONAL LABORATORY

@carlnotsagan

OVERVIEW

Co-processors, including GPUs

- Overview
- Memory management
- CPUs versus GPUs

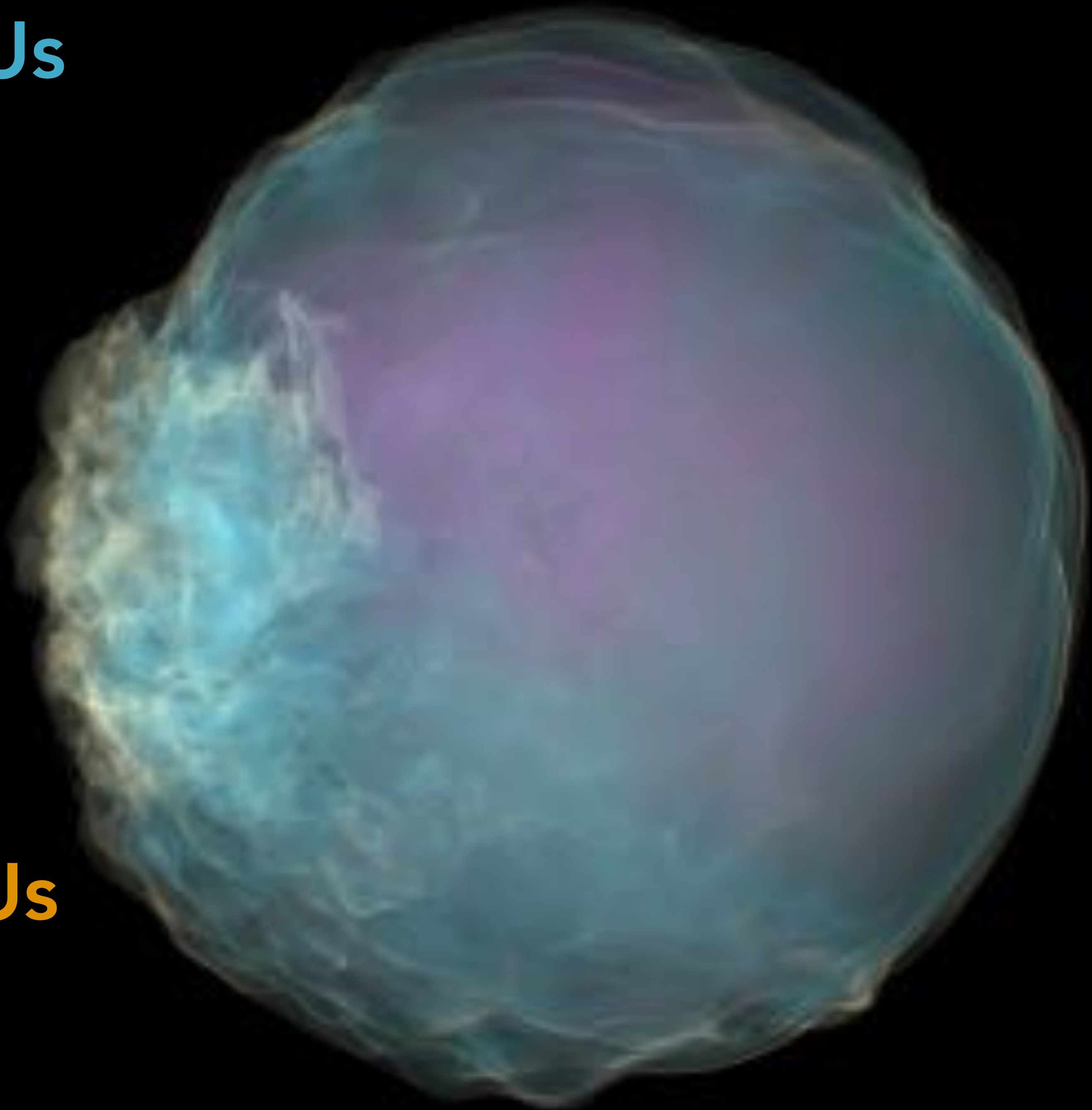
Utilizing modern GPUs

- *CUDA* programming model
- *CUDA* kernels
- GPU hierarchies

Parallel Computing with GPUs

- Numba
- cuPy
- cuDF

Summary



Oxygen shell burning in a 20 solar mass star.

CO-PROCESSORS, INCLUDING GPUS

Graphics Processing Unit (GPU) - Overview

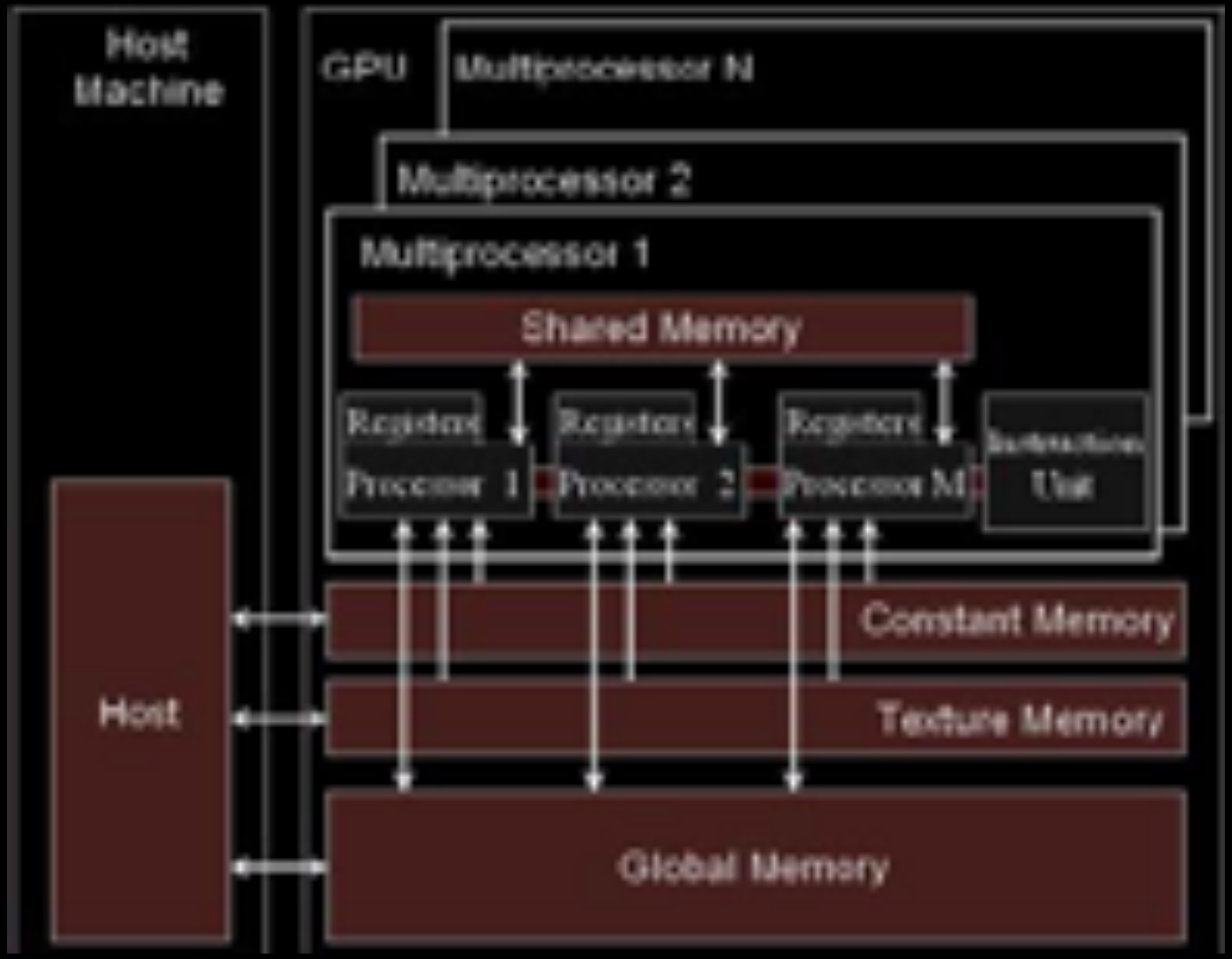
- designed for fast graphics processing
- graphics are a form of arithmetic
- have gradually evolved a design that is also useful for non-graphics computing
- Are not standalone, work alongside CPU (host) - coprocessor

NVIDIA Ampere A100 Tensor Core GPU is the world's most powerful accelerator for deep learning, machine learning, high-performance computing, and graphics.

CO-PROCESSORS, INCLUDING GPUS

GPUs - Anatomy of a GPU

- NVIDIA GPU often has 16-30 **Streaming Multiprocessors** (SMs).
- Each SM contains 8 **Streaming Processors** (SPs) - or processing cores.
- In general, many more cores but the cores are more limited than in CPU.

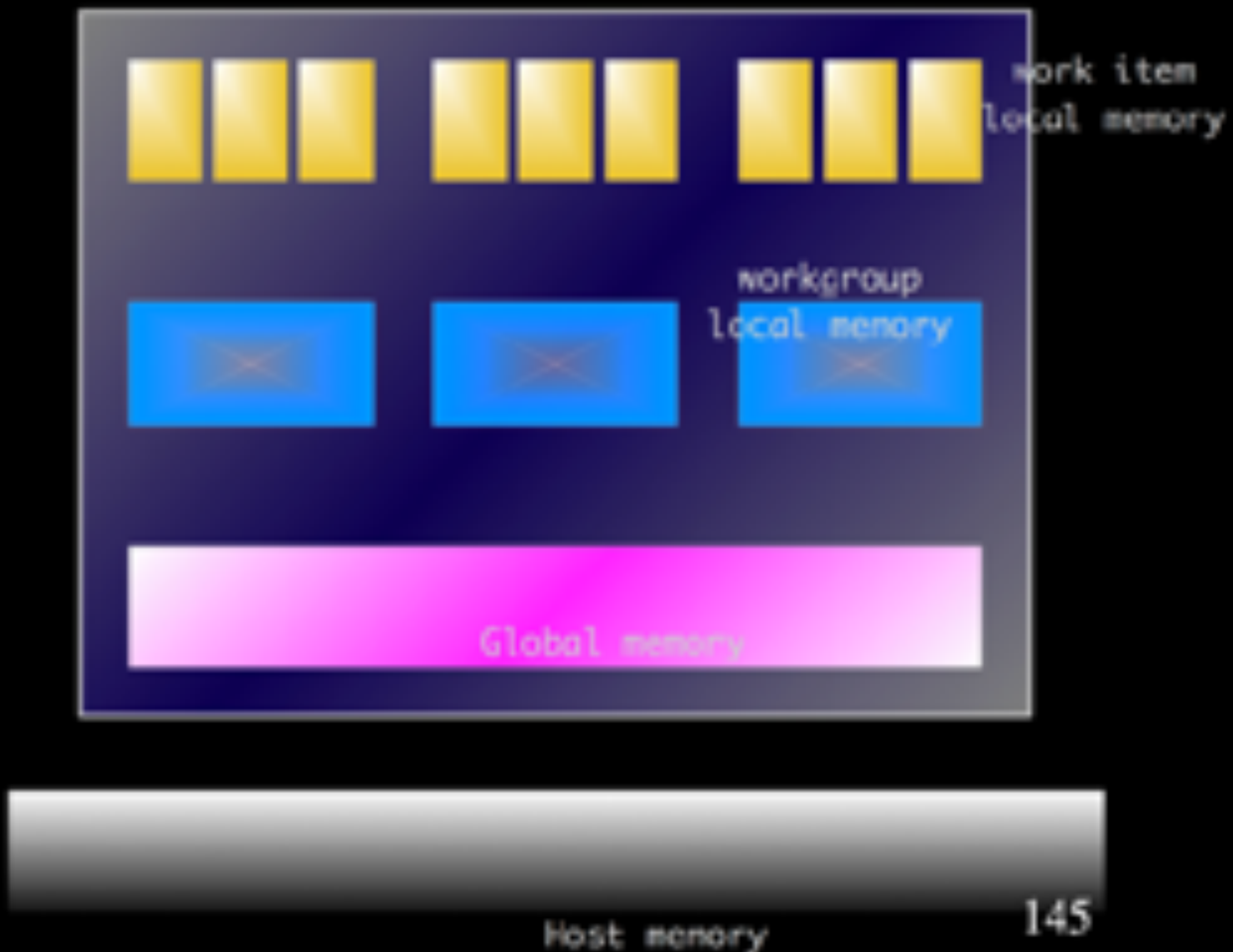


Qualitative diagram of a GPU.

CO-PROCESSORS, INCLUDING GPUS

GPUs - Memory Management

- In CPUs, we saw the memory hierarchy and different levels of cache.
- In *GPUs*, the solution is to support many more threads with fast switching between them.
- Because of this, memory management is key in GPU computing. Smaller caches!



Memory structure of a GPU.

CO-PROCESSORS, INCLUDING GPUS

GPUs - GPUs versus CPUs

CPUs:

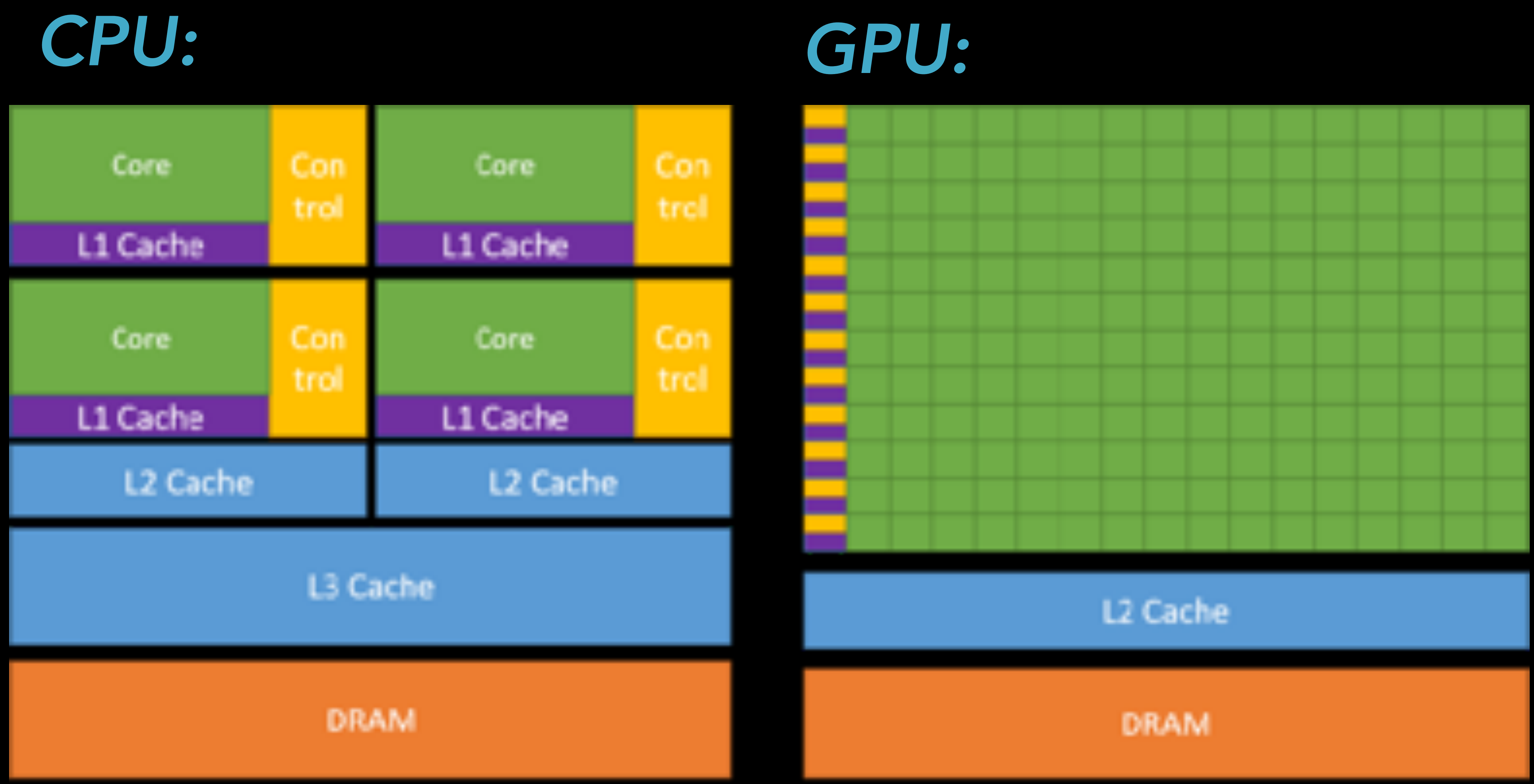
- Data exist on the CPU - somewhere.
- Peak performance = more cores. More flexible cache.
- Single stream of potentially very different instructions.
- Perform well on a single or few threads.

GPUs:

- GPUs are co-processors, meaning they require data to be transferred.
- Limited cache size, datatype (size) limited. Smaller datatypes = higher peak performance.
- Poor performance on "traditional codes"
- Designed to perform well on *many* threads.

CO-PROCESSORS, INCLUDING GPUS

GPUs - GPUs versus CPUs



Schematic diagram of CPU and GPU.

CO-PROCESSORS, INCLUDING GPUS

GPUs - Expected benefits

- Very good at doing **data parallel** computing
- CUDA provides a tool for writing code for the GPU
- Requires computation to have "enough data parallelism".
- Other co-processors exist!

Stampede 2 at Texas Advanced Computing Center. Uses Intel Knights Landing many-core processors as stand alone processors.

CO-PROCESSORS, INCLUDING GPUS

Your target architecture can determine your approach

UTILIZING MODERN GPUS

CUDA Programming Model

- CUDA®: A General-Purpose Parallel Computing Platform and Programming Model
- Comes with a software environment that allows developers to use C++ as a high-level programming language
- Automatic scalability to newer GPUs

UTILIZING MODERN GPUS

CUDA Programming Model - CUDA Kernels

- Functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

UTILIZING MODERN GPUS

CUDA Programming Model - Thread Hierarchy

- The **index of a thread** and its **thread ID** relate to each other in a straightforward way. In this example, the thread ID of a thread of index *(x, y)* is $(x + y\ D_x)$. Requires defining dimensions of thread blocks.

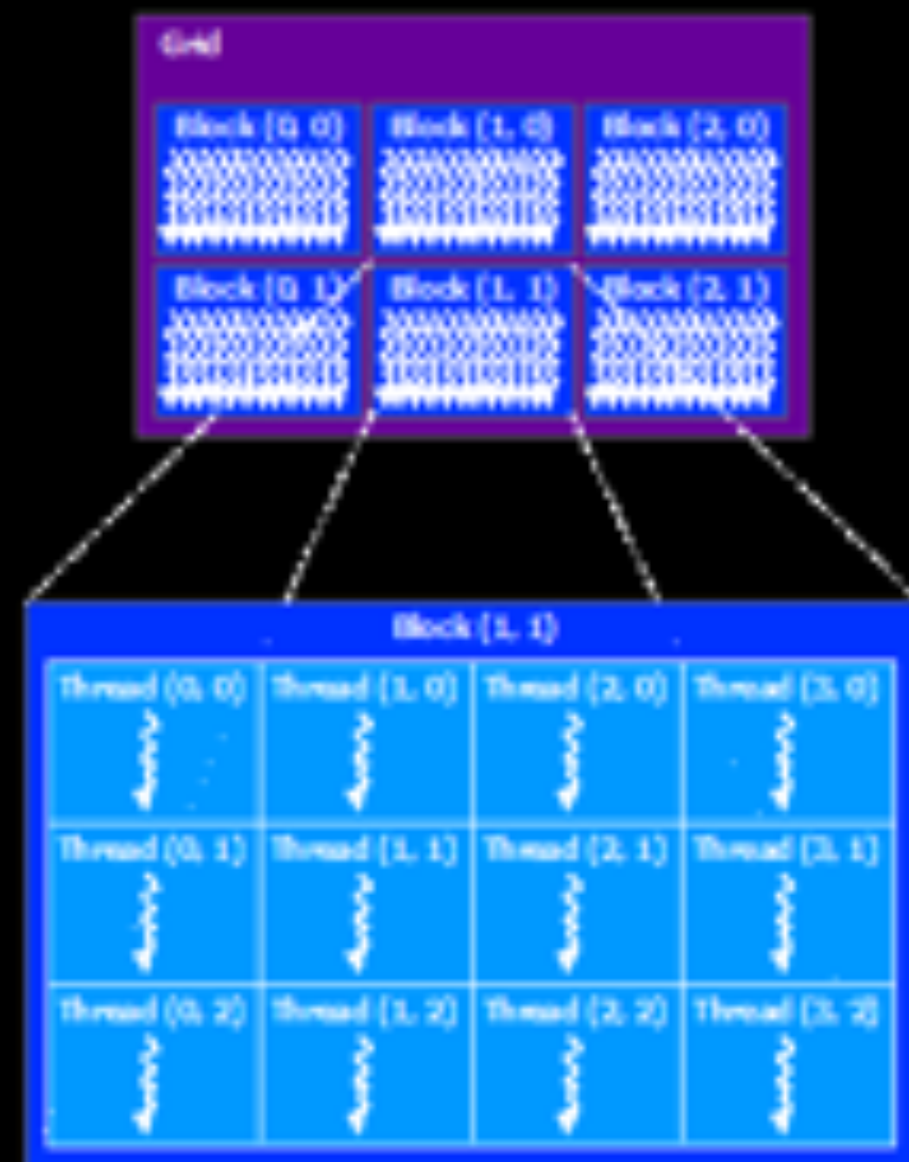
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

UTILIZING MODERN GPUS

CUDA Programming Model - Grids and Thread Blocks

- Kernels can be executed by individual **threads** or multiple equally-shaped thread blocks.
- **Blocks** are a collection of threads.
- A **Grid** is a collection of **Block**s.

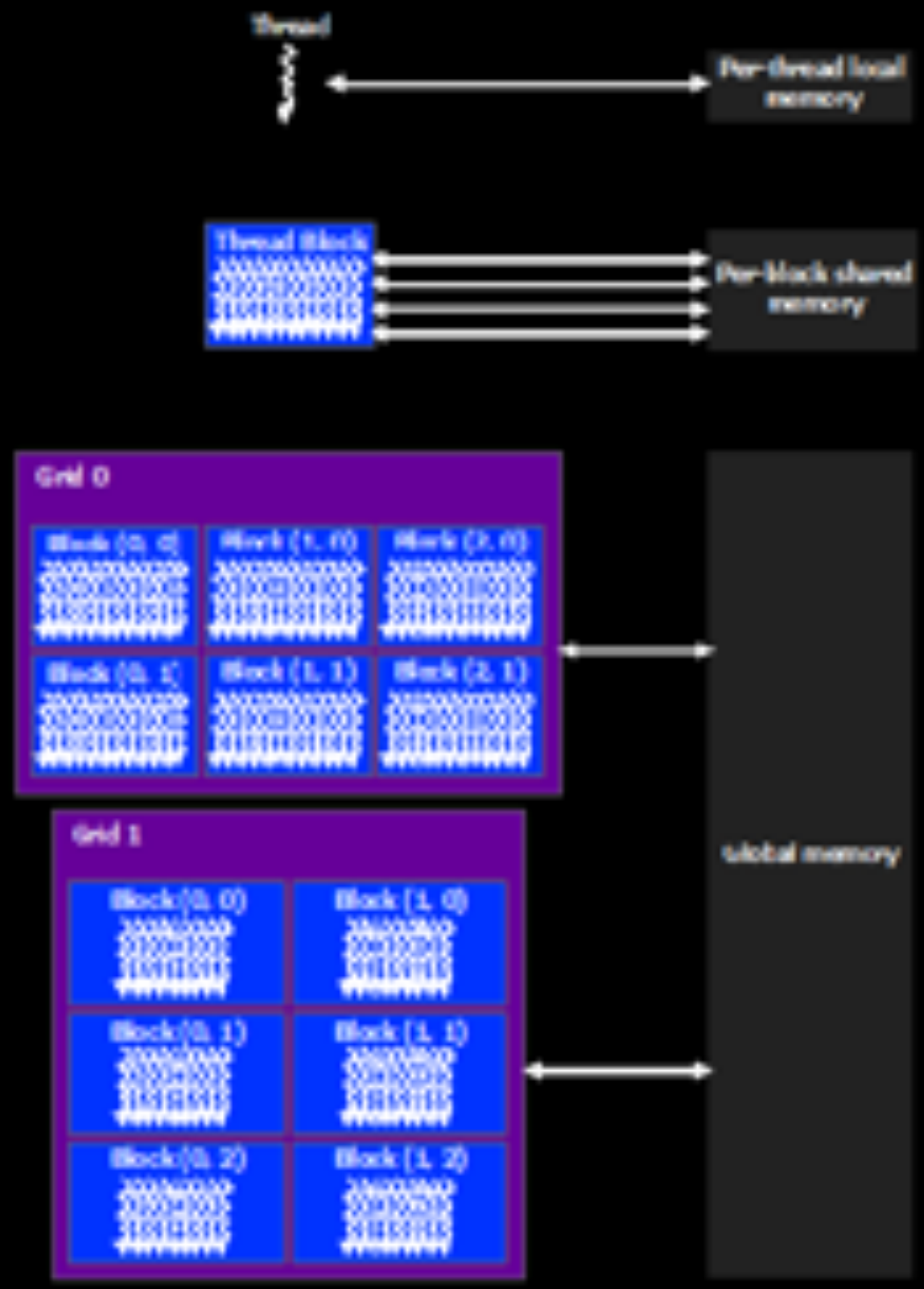


Schematic diagram of Thread Blocks

UTILIZING MODERN GPUS

CUDA Programming Model - Memory Hierarchy

- Threads have access to local memory.
- Blocks can share memory amon threads
- Grids have access to global memory.
- Programs designed with memo hierarchy in mind.



UTILIZING MODERN GPUS

CUDA Programming Model - Heterogeneous Computing

- CUDA model assumes GPUs operate as co-processors.
- Requires explicit management of data to and from **device.**
- CUDA Programming interface has many options including in Python!

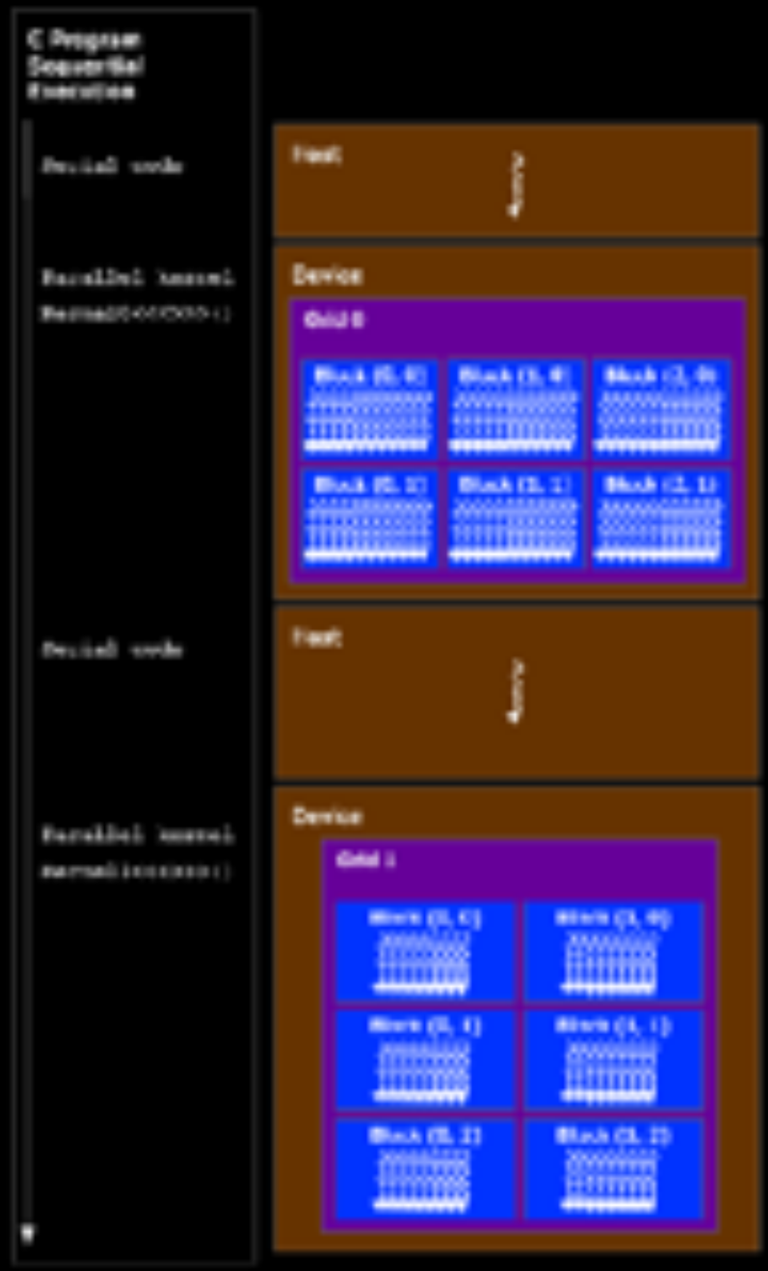


Diagram showing Heterogeneous Programming.

UTILIZING MODERN GPUS

Tools leveraging the CUDA programming model in Python exist!

PARALLEL COMPUTING WITH GPUS

Numba makes Python code fast (on GPUs too)

Kernel declaration:

```python
@cuda.jit
def increment_by_one(an_array):
    """
    Increment all array elements by one.
    """
    # code elided here; read further for different implementations
```

PARALLEL COMPUTING WITH GPUS

Numba makes Python code fast (on GPUs too)

Kernel invocation:

```
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1)) // threadsperblock
increment_by_one[blockspergrid, threadsperblock](an_array)
```

- Depends on array size, requires some knowledge of available threads

PARALLEL COMPUTING WITH GPUS

Numba makes Python code fast (on GPUs too)

Choosing the block size:

- On the software side, the block size determines how many threads share a given area of shared memory.
- On the hardware side, the block size must be large enough for full occupation of execution units.
- Code will typically run but not be maximally efficient tools for measuring efficiency and block size.

PARALLEL COMPUTING WITH GPUS

Numba makes Python code fast (on GPUs too)

Thread positioning:

```python
@cuda.jit
def increment_by_one(an_array):
    # Thread id in a 1D block
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    if pos < an_array.size:  # Check array boundaries
        an_array[pos] += 1
```

- Tools for determining thread positioning.

PARALLEL COMPUTING WITH GPUS

CuPy – NumPy & SciPy for GPU

- *CuPy* acts as a drop-in replacement to run existing NumPy/SciPy code on NVIDIA CUDA or AMD ROCm platforms
- CuPy provides a *ndarray*, sparse matrices, and the associated routines for GPU devices, all having the same API as NumPy and SciPy

PARALLEL COMPUTING WITH GPUS

CuPy – NumPy & SciPy for GPU

CuPy

```
>>> squared_diff = cp.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'squared_diff')
```

1. Define datatypes
2. Define equation
3. Name function call

Example - User Defined Kernel

PARALLEL COMPUTING WITH GPUS

CuPy – NumPy & SciPy for GPU

CuPy

```
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cp.arange(5, dtype=np.float32)
>>> squared_diff(x, y)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
>>> squared_diff(x, 5)
array([[25., 16.,  9.,  4.,  1.],
       [ 0.,  1.,  4.,  9., 16.]], dtype=float32)
```

1. Define similar to NumPy arrays.
2. Call equation

```
>>> z = cp.empty((2, 5), dtype=np.float32)
>>> squared_diff(x, y, z)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
```

1. Can also define output arrays.

Example - User Defined Kernel - Invocation

PARALLEL COMPUTING WITH GPUS

CuPy – NumPy & SciPy for GPU

- Supports Type-generic kernels
- Supports Reduction kernels
- Support Raw kernels and many more!

PARALLEL COMPUTING WITH GPUS

cuDF - GPU DataFrames

- cuDF is a Python GPU DataFrame library (built on the Apache Arrow columnar memory format) for loading, joining, aggregating, filtering, and otherwise manipulating data
- cuDF also provides a pandas-like AP
- Accelerate their workflows without going into the details of CUDA programming

PARALLEL COMPUTING WITH GPUS

cuDF - GPU DataFrames

RAPIDS

When to use what:

- workflow is fast enough on a single GPU or your data comfortably fits in memory on a single GPU, you would want to use **cuDF**
- want to distribute your workflow across multiple GPUs, have more data than you can fit in memory on a single GPU, or want to analyze data spread across many files at once, you would want to use **Dask-cuDF**.

Specific examples at https://docs.rapids.ai/api

The best approach will likely be a combination of different tools.

SUMMARY

Co-processors, including GPUs

- Anatomy and uses for GPUs
- Memory and thread hierarchy of GPUs
- Comparisons to CPUs

Utilizing modern GPUs

- *CUDA* programming model
- GPU hierarchies in *CUDA*
- Heterogeneous computing

Parallel Computing with GPUs (in Python)

- *Numba* - writing *CUDA* kernels
- *CuPy* - NumPy & SciPy for GPU
- *cuDF* - GPU DataFrames

THANK YOU

Worry about the data,
operational intensity,
(and memory hierarchy)!

Web: ***carlnotsagan.com***
Email: ***carlnotsagan@lanl.gov***

Los Alamos
NATIONAL LABORATORY

@carlnotsagan