

“The manufacturers quote a peak performance of 28 GFLOPS for the largest CM-2 (65,536 PEs) with floating point accelerators”

CM-2 evaluation, 1987

Lecture B.3: Accelerated Computing

CS205: Computing Foundations for Computational Science
Dr. David Sondak
Spring Term 2021



HARVARD
School of Engineering
and Applied Sciences



IACS INSTITUTE FOR APPLIED
COMPUTATIONAL SCIENCE
AT HARVARD UNIVERSITY

Lectures developed by Dr. Ignacio M. Llorente

Before We Start

Where We Are

Computing Foundations for Computational and Data Science

How to use modern computing platforms in solving scientific problems

Intro: Large-Scale Computational and Data Science

A. Parallel Processing Fundamentals

B. Parallel Computing

B.1. Foundations of Parallel Computing

B.2. Performance Optimization

B.3. Accelerated Computing

B.4. Shared-memory Parallel Processing

B.5. Distributed-memory Parallel Processing

C. Parallel Data Processing

Wrap-Up: Advanced Topics

CS205: Contents

APPLICATION SOFTWARE

APPLICATION
PARALLELISM

PARALLEL PROGRAM
DESIGN



Optimization

PROGRAMMING MODEL

OpenACC

Spark

OpenMP

Map-Reduce

MPI

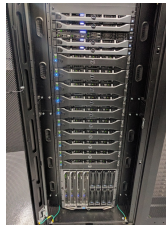
B. BIG COMPUTE

PLATFORM

C. BIG DATA



CLOUD COMPUTING



PARALLEL ARCHITECTURES

Context

Accelerated Computing

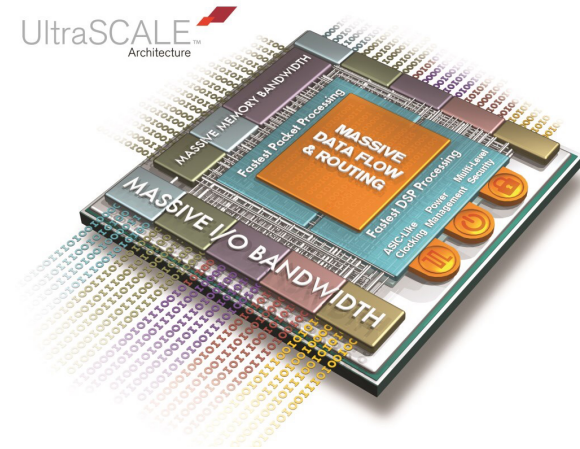
GPU-Accelerated Computing

- Enable high degree of parallelism – each GPU has thousands of cores
- Consistent, well documented APIs (OpenACC, CUDA and OpenCL) and tools
- Widely supported by vendors and OSS project



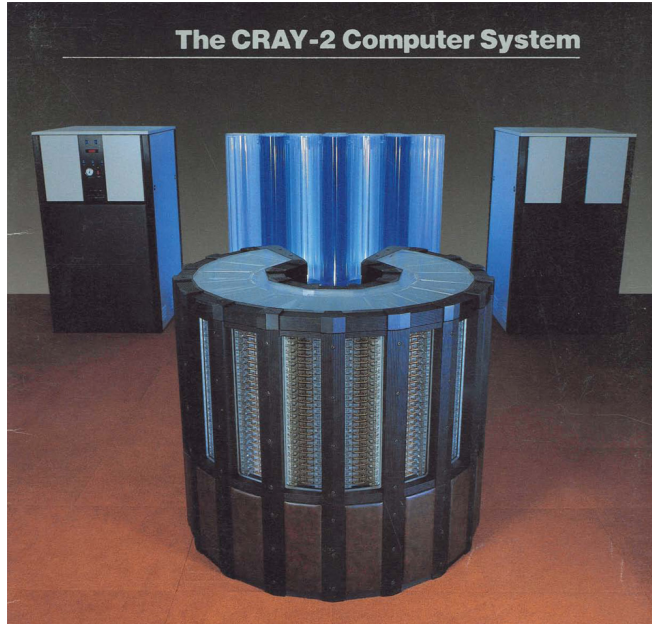
FPGA-Accelerated Computing

- Massive parallel – each FPGA includes millions of parallel logic cells
- Flexible – lower level, hardware, no fixed instruction set
- Programmable using FPGA development tools

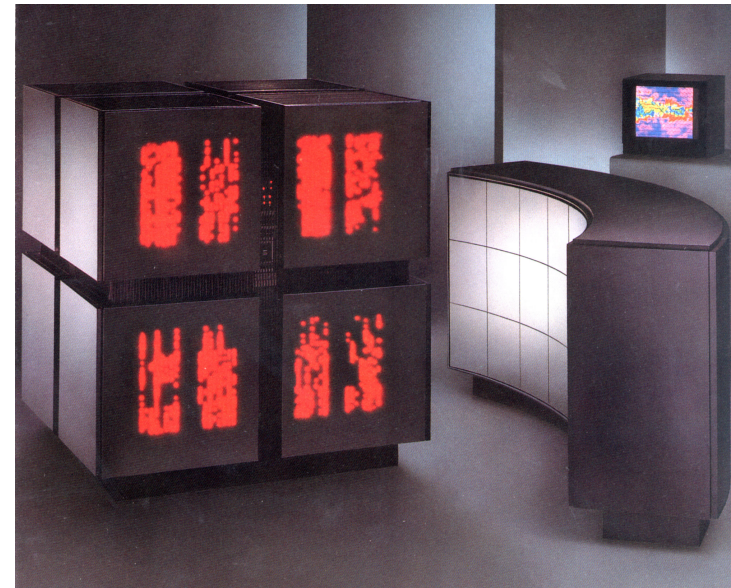


Context

Really New?: Thinking Machines CM-2 (1990)



Vector



SIMD

Context

Top Hollywood Computer: Thinking Machines CM-2 (1990)



Context

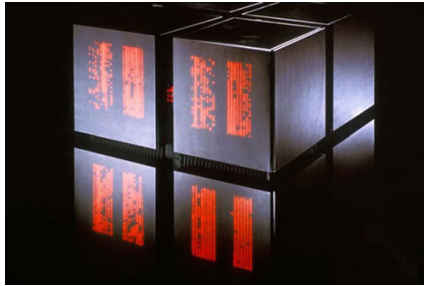
Really New? – NVIDIA GEFORCE RTX (2021)

	GEFORCE RTX 3090	GEFORCE RTX 3080	GEFORCE RTX 3070	GEFORCE RTX 3060 TI	GEFORCE RTX 3060
GPU Engine Specs:					
NVIDIA CUDA® Cores	10496	8704	5888	4864	3584
Boost Clock (GHz)	1.70	1.71	1.73	1.67	1.78
Base Clock (GHz)	1.40	1.44	1.50	1.41	1.32
Memory Specs:					
Standard Memory Config	24 GB GDDR6X	10 GB GDDR6X	8 GB GDDR6	8 GB GDDR6	12 GB GDDR6
Memory Interface Width	384-bit	320-bit	256-bit	256-bit	192-bit
Technology Support:					
Ray Tracing Cores	2nd Generation	2nd Generation	2nd Generation	2nd Generation	2nd Generation
Tensor Cores	3rd Generation	3rd Generation	3rd Generation	3rd Generation	3rd Generation
NVIDIA Architecture	Ampere	Ampere	Ampere	Ampere	Ampere




Context

SIMD Converted into a Commodity



2-4 Millions of Dollars (2018)



NVIDIA GEFORCE RTX 3090

- > Cooling System: Fan
- > Boost Clock Speed: 1.70 GHz
- > GPU Memory Size: 24 GB

\$1,499.⁹⁹

CHECK AVAILABILITY

[+ Compare](#)

Roadmap

Performance Optimization and Accelerators

Heterogeneous Computing

GPU Computing

GPU Programming

OpenACC Fundamentals

HETEROGENEOUS COMPUTING

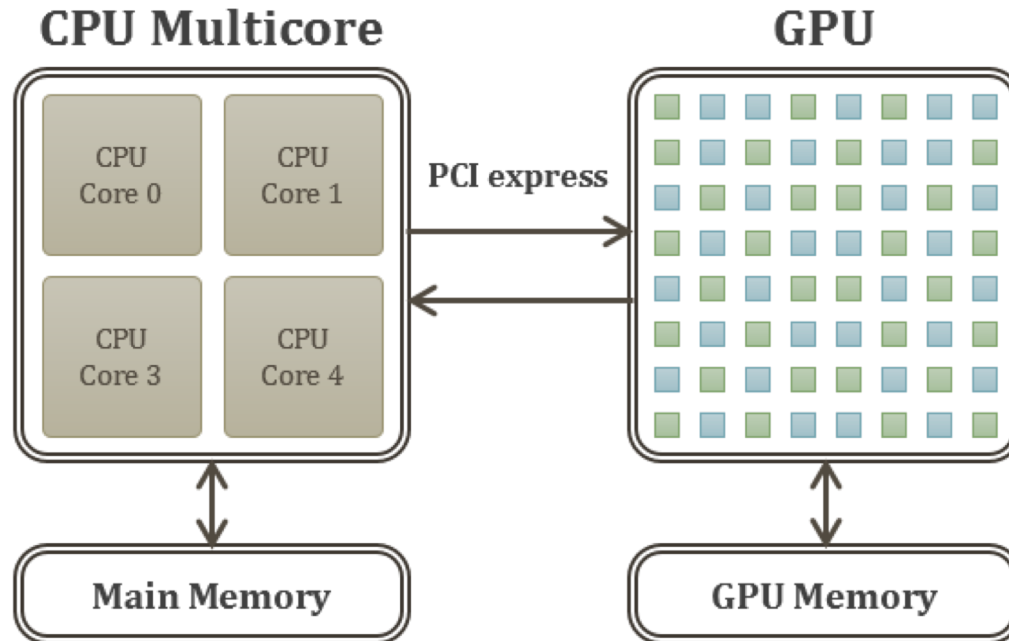
Heterogeneous Computing

Intersection of CPU and GPU Computing

- Heterogeneous computing refers to **systems that use more than one kind of processor**
- Multi-core systems gain performance not just by adding cores, but also by **incorporating specialized processing capabilities** to handle particular task
- Heterogeneous System Architecture (HSA) systems utilize multiple processor types (typically CPUs and GPUs), usually on the same silicon die, to give you the best of both worlds:
 - ✓ While CPUs can run the operating system and perform traditional serial or multi-threading tasks
 - ✓ GPUs have vector processing capabilities that enable them to perform parallel operations on very large sets of data – and to do it at much lower power consumption relative to the serial processing of similar data sets on CPUs

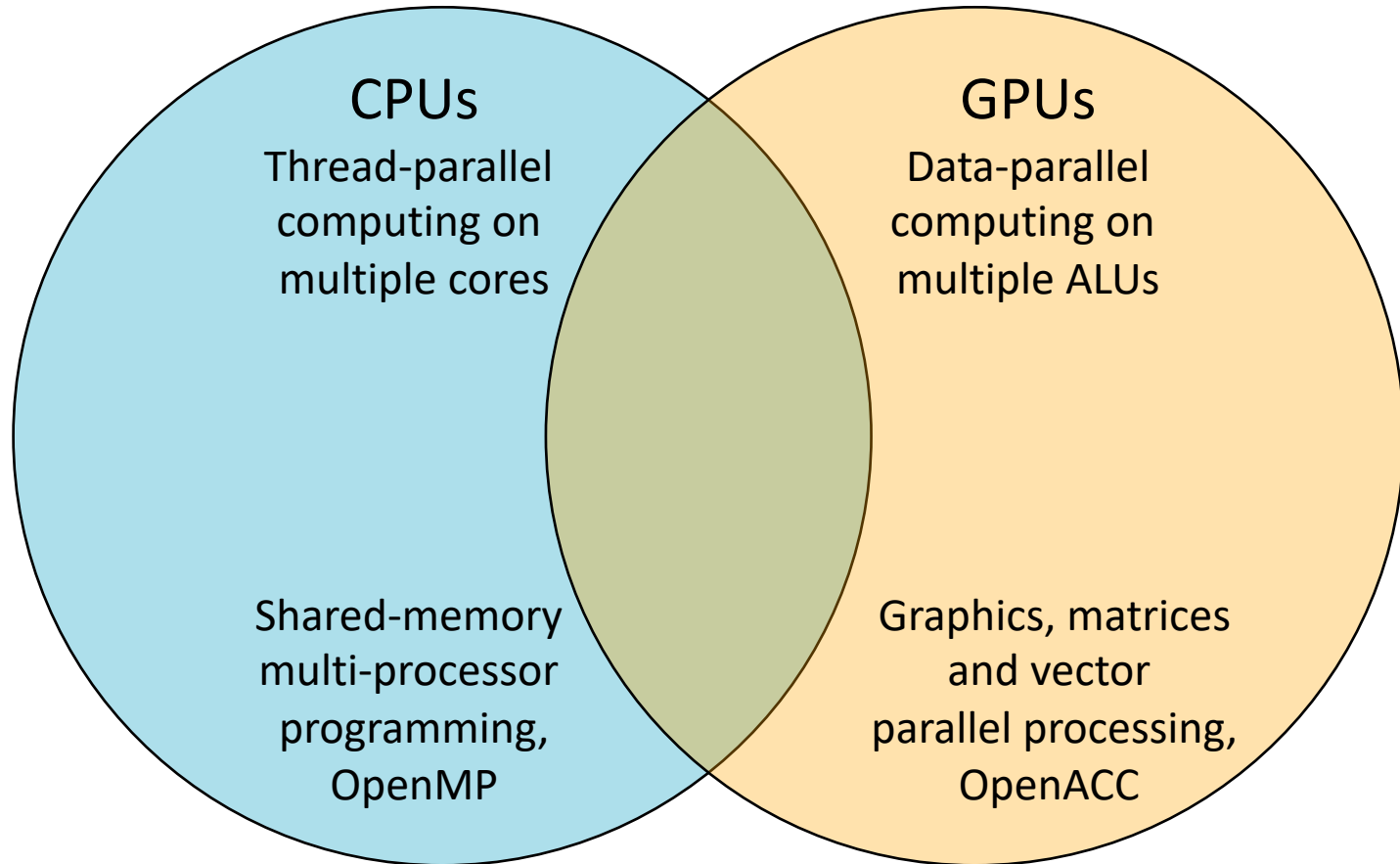
Heterogeneous Computing

Intersection of CPU and GPU Computing



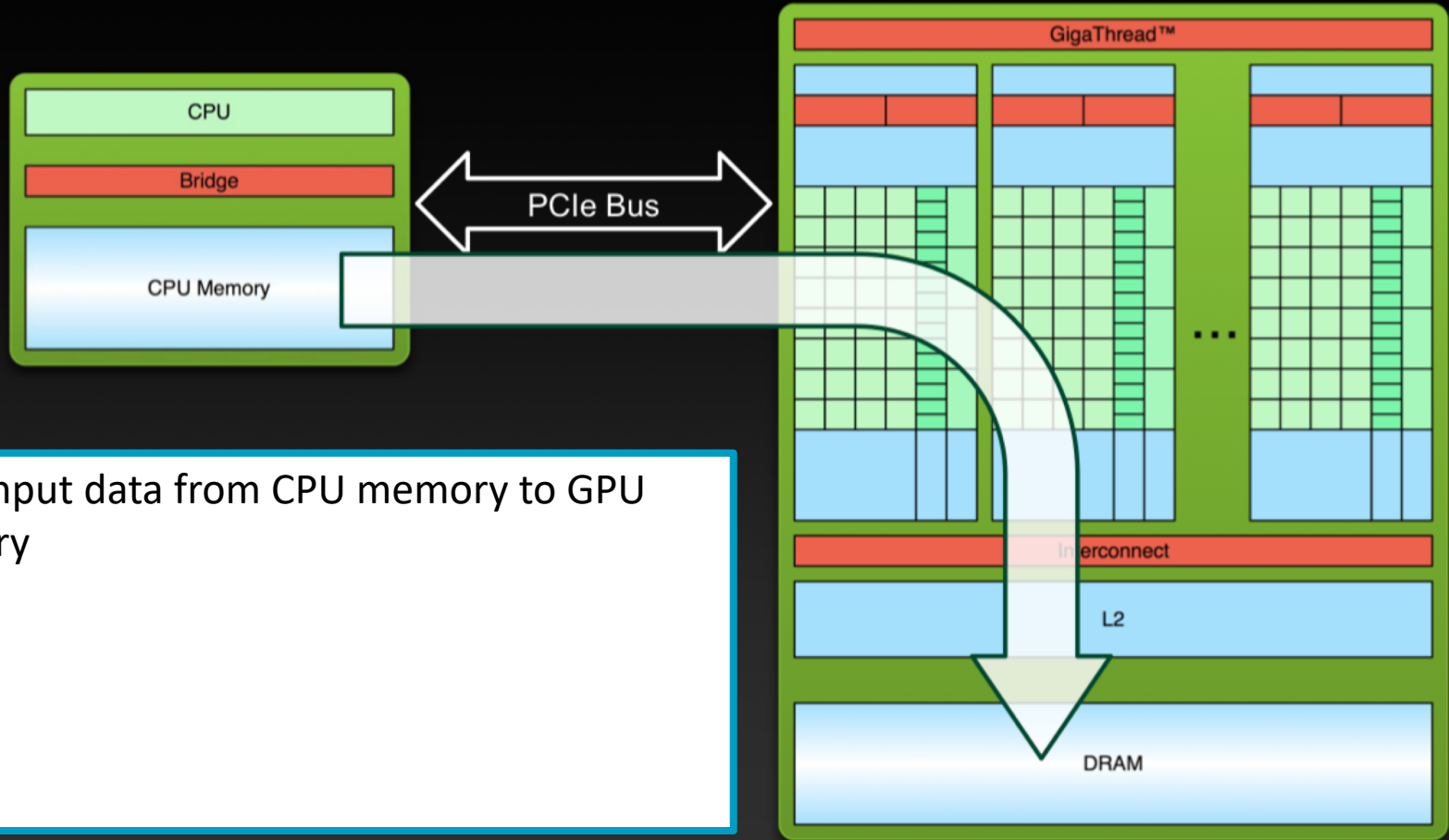
Heterogeneous Computing

Intersection of CPU and GPU Computing



Heterogeneous Computing

Execution Model

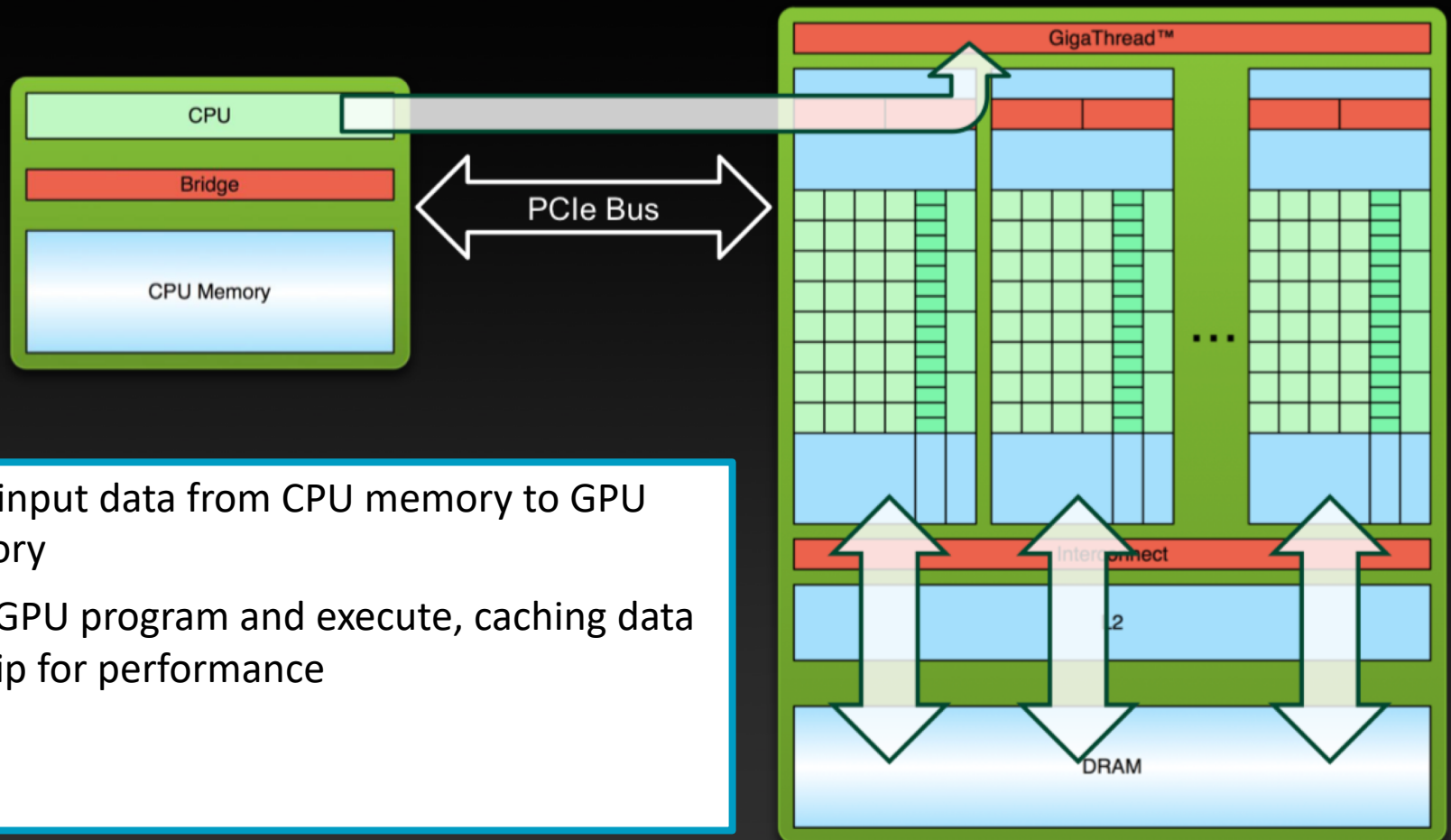


1. Copy input data from CPU memory to GPU memory

Source: NVIDIA

Heterogeneous Computing

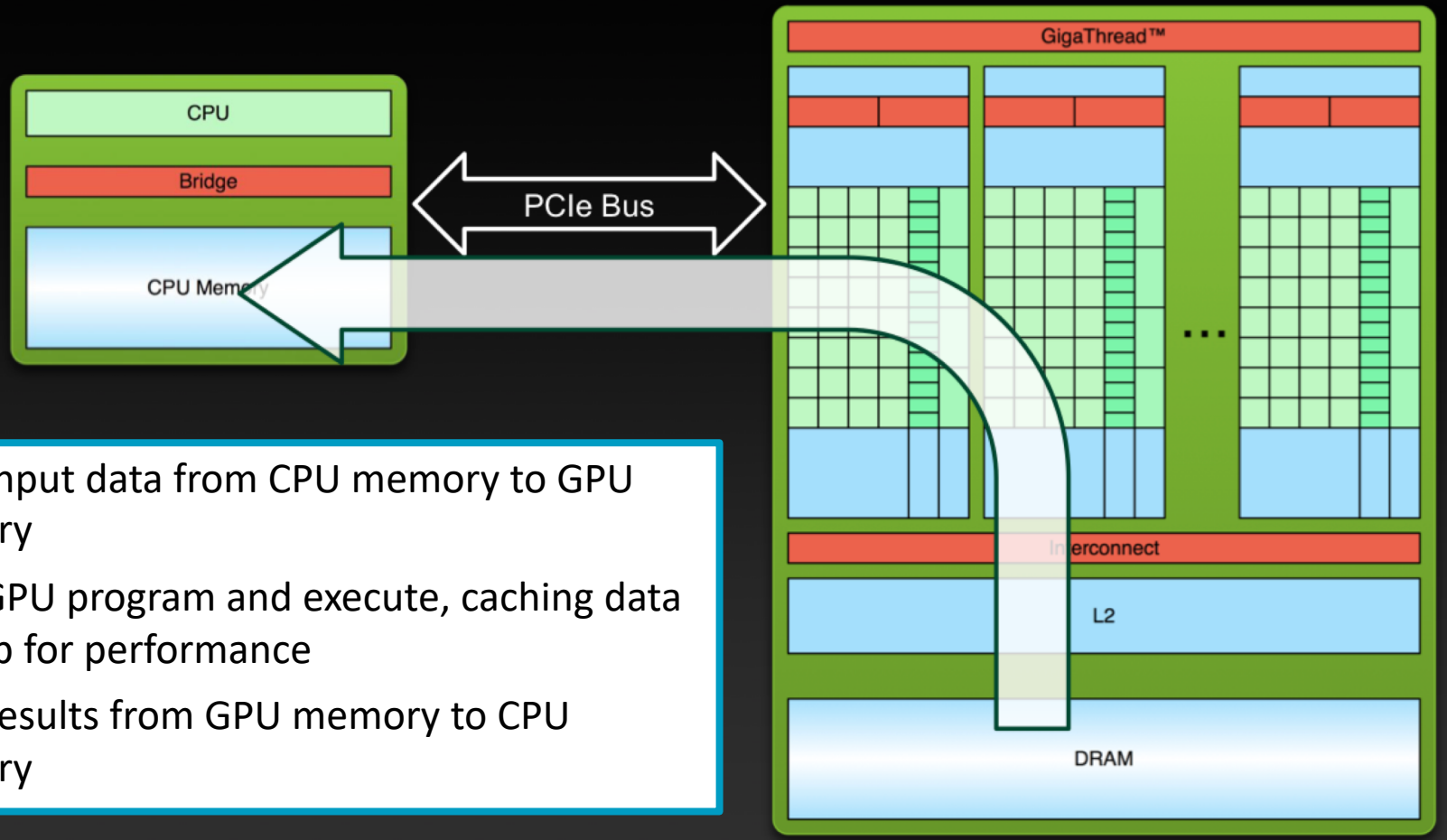
Execution Model



Source: NVIDIA

Heterogeneous Computing

Execution Model



Source: NVIDIA

GPU COMPUTING

GPU Computing

What is it?

- GPU-accelerated computing is the use of a graphics processing unit (GPU) together with a CPU to accelerate deep learning, analytics, and engineering applications (data parallel applications)
- GPU-accelerated computing offloads compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU. From a user's perspective, applications simply run much faster

- The Graphic Processing Unit (GPU) is a processor that was specialized for processing graphics
- The GPU has evolved towards a more flexible architecture.
 - ✓ Opportunity: We can implement *any algorithm*, not only graphics
 - ✓ Challenge: obtain efficiency and high performance
- The idea is to obtain performance through SIMD
 - ✓ Same operations (**kernel**) on multiple data

GPU Computing

Main Downside

Only for GPU-friendly (Data Parallel) Applications

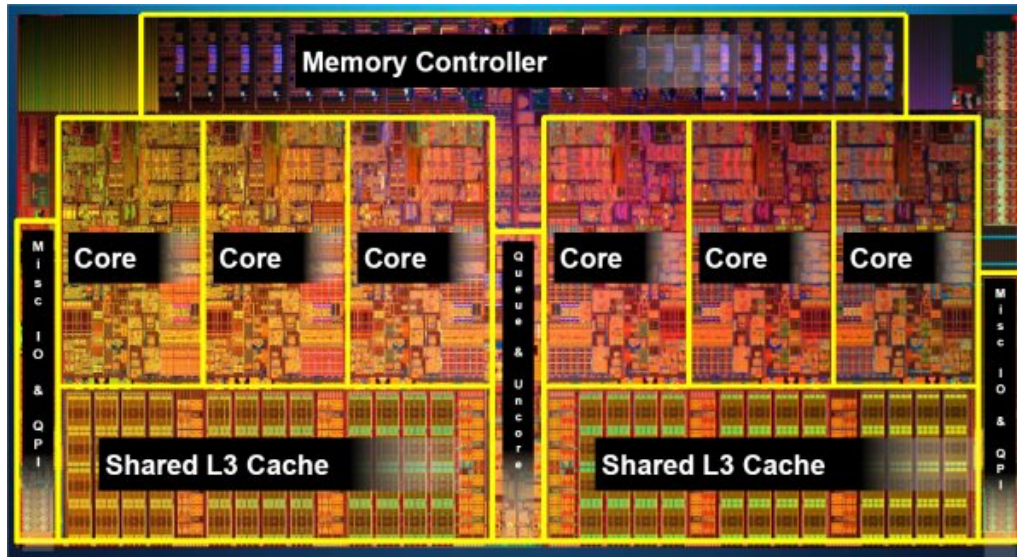
- Computer graphics
- Texture, rendering, image processing...
- Matrix operations
- Structured simulations (finite differences)

Downsides

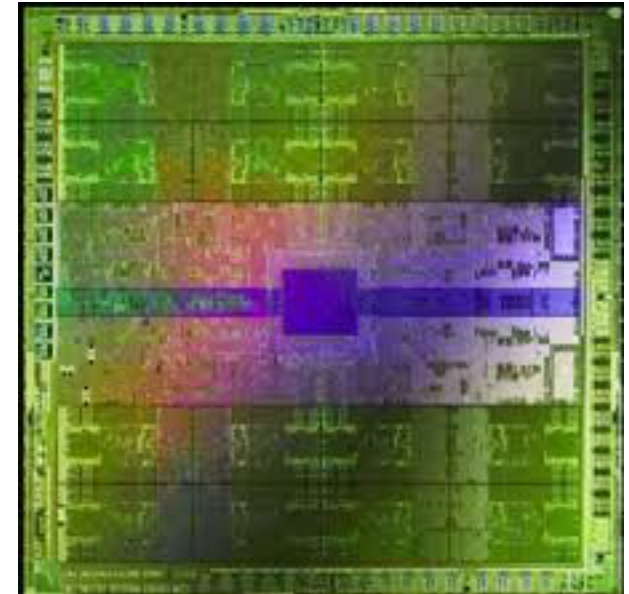
- Not-general purpose CPUs
- Difficult to program
- Difficult to tune: Bandwidth vs. Compute vs. Context
- CPU-GPU link has been slow, historically (system bus)

GPU Computing

An Alternative Way to Use the Transistors



Intel i7 980x (Extreme)
6 cores
1.2B transistors



NVIDIA GTX 580 SC
512 cores
3B transistors

Cache and memory hierarchy vs more cores and ALUs

Optimized for low-latency access to cache
Complex control logic for ILP

Optimized for data-parallel, throughput
computation

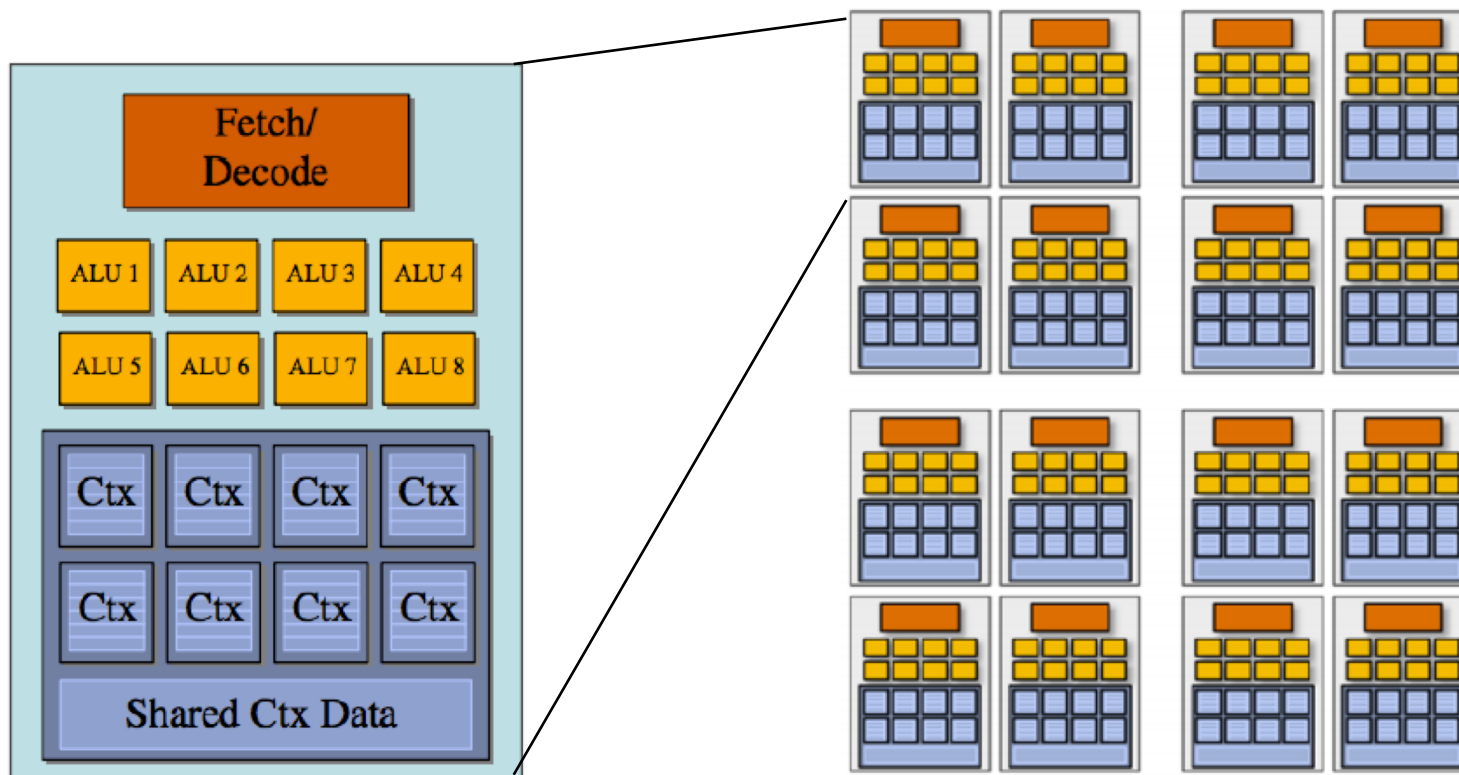
More transistors for computation

GPU Computing

Better Performance/Cost for Regular Data Processing

Add SIMD Processing on Many Cores

- Amortize cost/complexity of managing an instruction stream across many ALUs

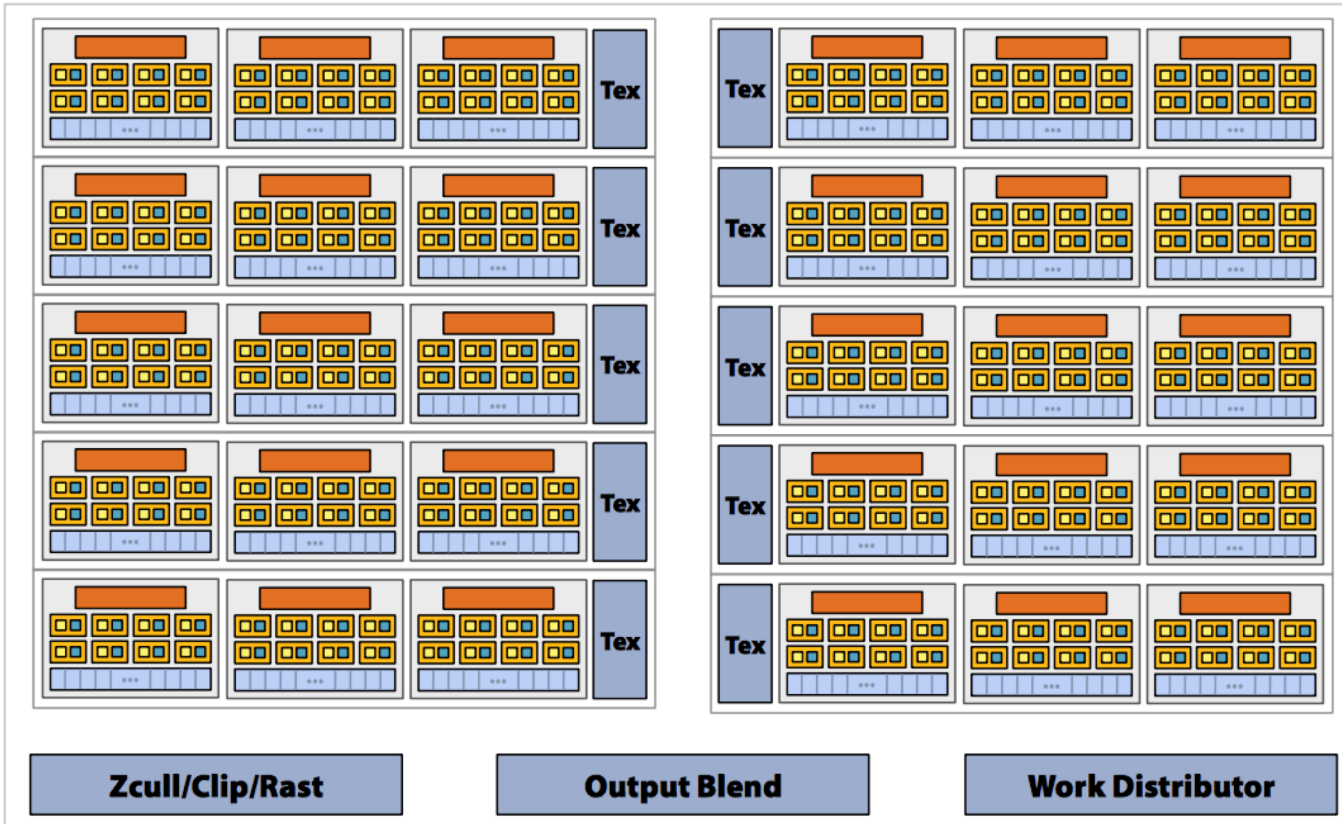


16 cores x 8 ALUs/core = 128 ALUs (mul-add) => 256 GFLOPs @1GHz

GPU Computing

Examples

Example: NVIDIA GTX 280

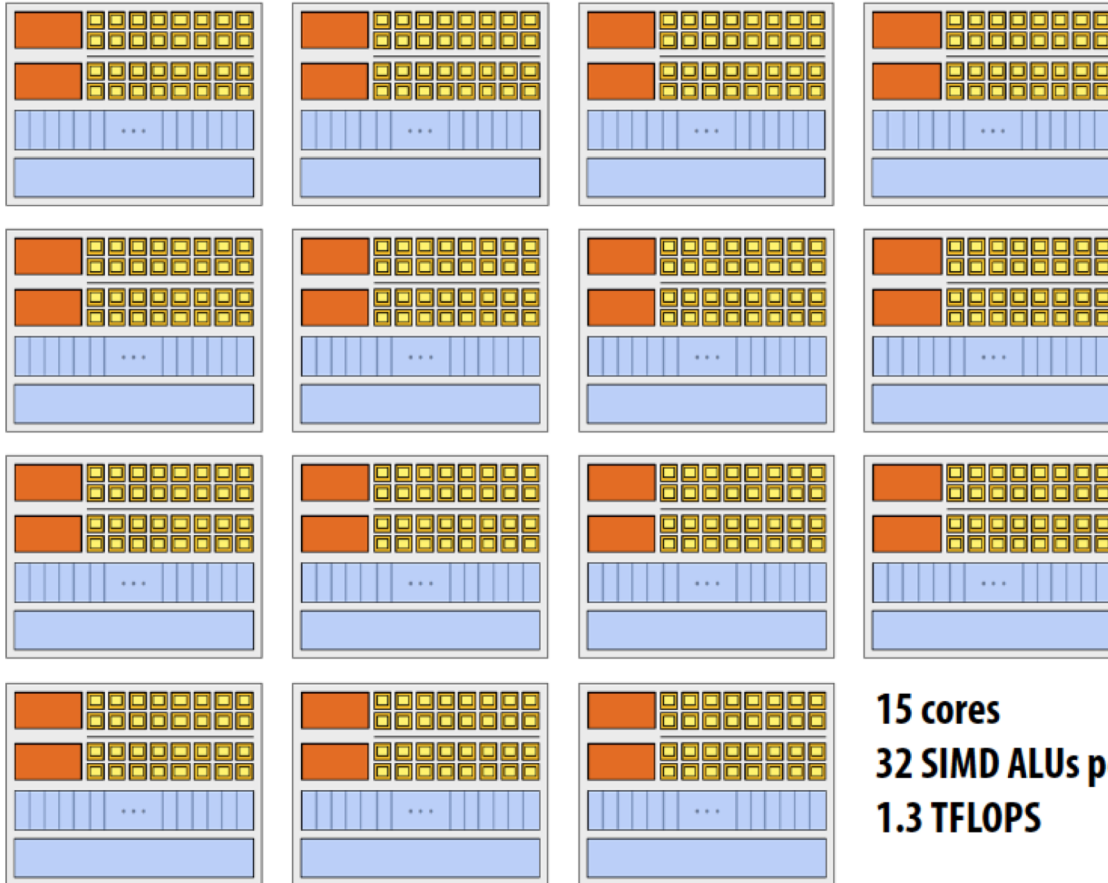


$30 \text{ cores} \times 8 \text{ ALUs/core} = 240 \text{ ALUs (3 FLOPS)} \Rightarrow 933 \text{ GFLOPs @1.3GHz}$

GPU Computing

Examples

Example: NVIDIA GTX 480



$15 \text{ cores} \times 32 \text{ ALUs/core} = 480 \text{ ALUs (4 FLOPS)} \Rightarrow 1.3 \text{ TFLOPs @0.7GHz}$

GPU Computing

Examples

Powered by NVIDIA Tesla M60 GPUs

- Each GPU supports 8 GiB of GPU memory, 2048 parallel processing cores

	Tesla M60	Tesla M6
Number of GPUs	2 NVIDIA Maxwell™ GPUs	1 NVIDIA Maxwell™ GPU
Total NVIDIA CUDA® Cores	4,096 (2048 per GPU)	1,536
Total Memory Size	16 GB GDDR5 (8 GB per GPU)	8 GB GDDR5
Max Power	300 W	100 W
Form Factor	PCIE 3.0 Dual Slot	MXM
Board Dimensions	10.5" x 4.4"	3.2" x 4.1"
Cooling Solution	Passive/Active	Bare Board

GPU PROGRAMMING

GPU Programming

Main Ideas

- Massively parallel with hundreds of cores and thousands of threads
- Loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
- Help Compiler:
 - ✓ It must be able to figure out sizes of data regions
 - ✓ Pointer arithmetic should be avoided if possible
 - ✓ Function calls within accelerated region must be in-lineable.

GPU Programming

Different Libraries and Approaches

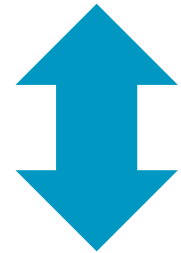
OpenACC
High level of abstraction

OpenCL
Device independent, but still requires data decomposition, transfer and synchronization

CUDA
Vendor/device dependent, use of explicit shared memory



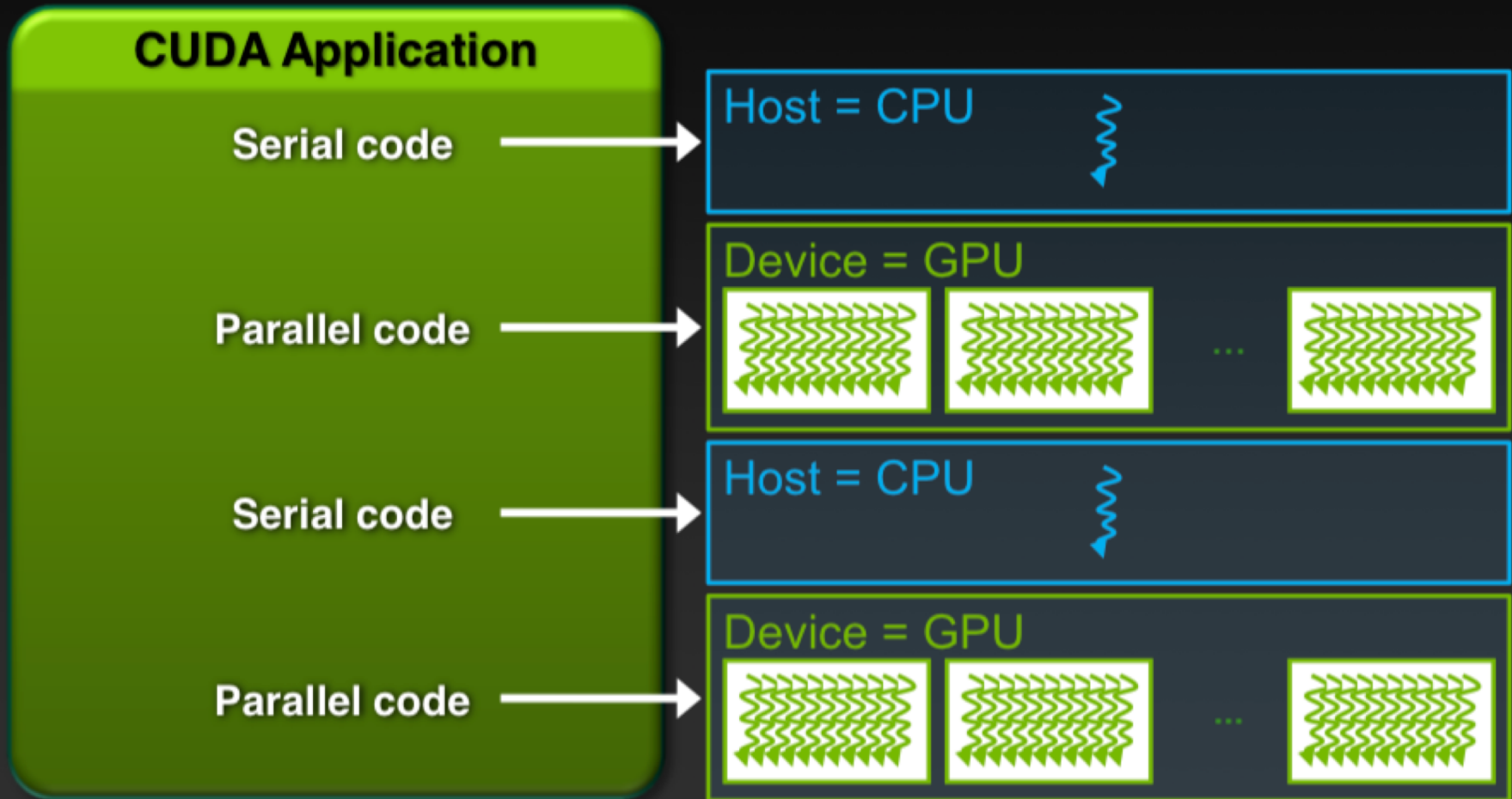
SIMPLICITY
PORTABILITY



PERFORMANCE
FUNCTIONALITY

GPU Programming

Anatomy of an Application



Source: NVIDIA

GPU Programming

Anatomy of an Application

CUDA

```
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

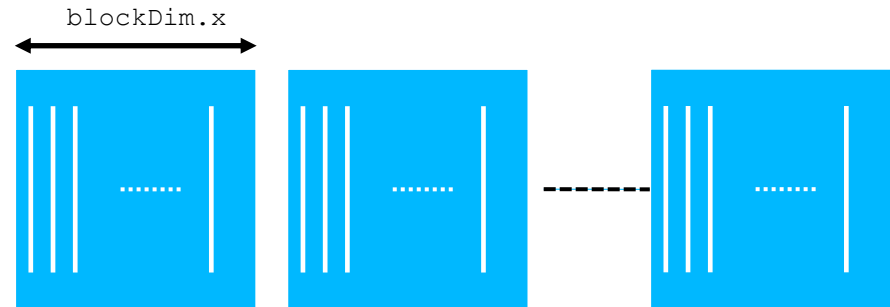
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY on 1M elements
    saxpy<<<(N+255)/256, 2.0f, d_x, d_y>>>(N, 2.0f, d_x, d_y);

    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
    ...
}
```

An Example: SAXPY with CUDA



Accelerated Computing

Anatomy of an Application

OpenACC

An Example: SAXPY with CUDA

```
#include <stdio.h>
#define vl 256
int main(void)
{
    int N = 1<<20;
    float *x, *y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    #pragma acc parallel vector_length(vl)
    for (int i = 0; i < N; i++) {
        y[i] = a*x[i] + y[i];
    }
    ...
}
```




**OpenACC is not
*GPU Programming.***

**OpenACC is
*Expressing Parallelism
in your code.***

OPENACC FUNDAMENTALS

OpenACC Fundamentals

An Open Specification

OpenACC

More Science, Less Programming



[About](#) [Tools](#) [News](#) [Stories](#) [Events](#) [Resources](#) [Spec](#) [Community](#)

What is OpenACC?

OpenACC is a user-driven directive-based performance-portable parallel programming model designed for scientists and engineers interested in porting their codes to a wide-variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a low-level model.

[Get Started](#)

or [take the next steps](#)

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    #pragma acc kernels {
    #pragma acc loop independent collapse(2)
    for ( int j = 1; j < n-1; j++ ) {
        for ( int i = 1; i < m-1; i++ ) {
            Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                                A [j-1] [i] + A [j+1] [i]);
            error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
        }
    }
}
}
```

OpenACC Fundamentals

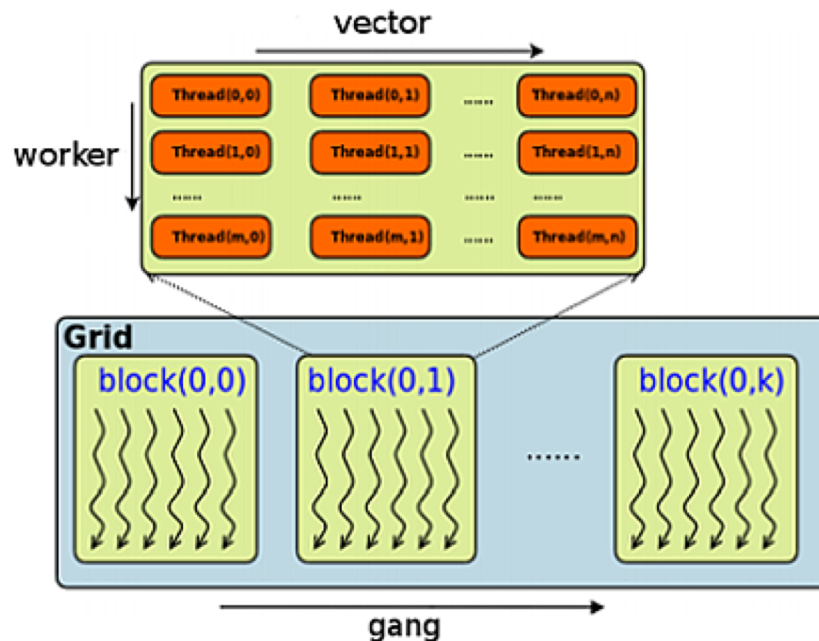
Why OpenACC versus CUDA or OpenCL?

- High-level. No involvement of OpenCL, CUDA, etc.
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial, non-GPU programmers can play along.
- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be quick.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms.
- Performance portable. Supports GPU accelerators and co-processors from multiple vendors, current and future versions.

OpenACC Fundamentals

Execution Model

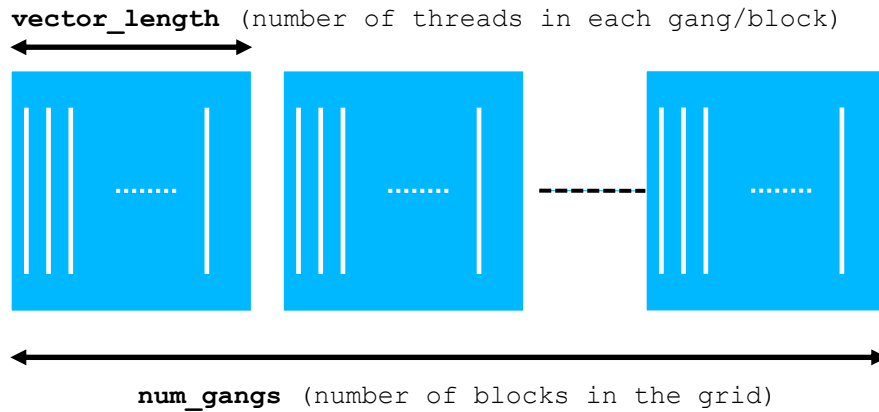
- The execution model has three parallelism levels: gang, worker, and vector
- A **gang** is comprised of one or multiple **workers**. All workers within a gang can share resources, such as memory or processor. Multiple gangs run completely independently.
- A **worker** computes one vector.
- **Vector** threads perform a single operation on multiple data (SIMD) in a single step.



OpenACC Fundamentals

Execution Model

- How these constructs map to the underlying hardware depends on the device capabilities and what the compiler thinks is the best mapping for the problem (tuning!)
- Mapping to NVIDIA GPUs
 - `gang==block`, `worker==warp`, and `vector==threads of a warp`
 - Usually omit “`worker`” and just have `gang==block`, `vector==threads of a block`



Tuning

`worker` is architecture-specific (32), related to thread scheduling group

`vector` should be multiple of 32 (usually 256/512)

OpenACC

Programming Model

- Compiler directives specify parallel regions (similar to OpenMP!)
- OpenACC compilers handle data between host and accelerators
- Intent is to be Portable (Independent of OS, CPU/accelerator vendor...)
- High-level programming: accelerator and data transfer abstraction

```
#pragma acc directive [clause [[, clause]]...]
{ structured block }
```

Parallel Construct => Generate one/several parallel gangs with same code

```
#pragma acc parallel [clause [[, clause]]...]
```

Loop Constructs => Generate parallel version of the loop

```
#pragma acc loop [clause [[, clause]]...]
```

Data Constructs => Defines explicit data transferring

```
#pragma acc data [clause [[, clause]]...]
```

OpenACC Fundamentals

A Simple Example: Pi

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
    double pi = 0.0f;
    long long i;
    #pragma acc parallel vector_length(vl)
    #pragma acc loop reduction(+:pi)
    for (i=0; i<N; i++) {
        double t= (double)((i+0.5)/N);
        pi +=4.0/(1.0+t*t);
    }
    printf("pi=%11.10f\n",pi/N);
    return 0;
}
```


OpenACC Fundamentals

A Simple Example: Pi

```
#include <stdio.h>
#define N 2000000000
#define vl 1024
int main(void) {
    double pi = 0.0f;
    long long i;
    #pragma acc parallel loop reduction(+:pi) vector_length(vl)
    for (i=0; i<N; i++) {
        double t= (double)((i+0.5)/N);
        pi +=4.0/(1.0+t*t);
    }
    printf("pi=%11.10f\n",pi/N);
    return 0;
}
```

OpenACC Fundamentals

Kernel Directive

- Identifies a region of code that may contain parallelism, but relies on the automatic parallelization capabilities of the compiler to analyze the region
- Developers with little or no parallel programming experience, or those working on functions containing many loop nests that might be parallelized will find the kernels directive a good starting place for OpenACC acceleration

```
#pragma acc kernels [clause ...]  
  { structured block }
```

Clauses

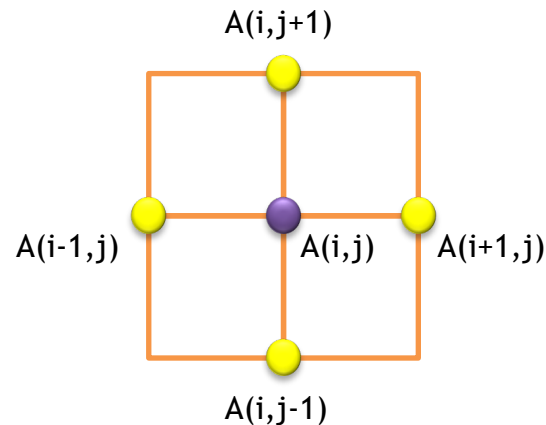
```
  if( condition )  
  async( expression )  
  any data clause
```

```
// Compute matrix multiplication  
#pragma acc kernels copyin(a,b) copy(c)  
  for (i = 0; i < SIZE; ++i) {  
    for (j = 0; j < SIZE; ++j) {  
      for (k = 0; k < SIZE; ++k) {  
        c[i][j] += a[i][k] * b[k][j];  
      }  
    }  
  }
```

OpenACC Fundamentals

Example: Jacobi Iteration

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
 - ✓ Common, useful algorithm
 - ✓ Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$



$$A_{k+1}(i, j) = \frac{A_k(i-1, j) + A_k(i+1, j) + A_k(i, j-1) + A_k(i, j+1)}{4}$$

OpenACC Fundamentals

Example: Jacobi Iteration

Iterate Across
Elements



Swap Input/Output
Arrays



```
while ( error > tol && iter < iter_max ) {
    error=0.0;

    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```

OpenACC Fundamentals

Example: First Try - Jacobi Iteration

Iterate Across
Elements

Swap Input/Output
Arrays

```
while ( error > tol && iter < iter_max ) {
    error=0.0;

    #pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```

OpenACC Fundamentals

Basic Concepts

Excessive Data Transfers

```
while ( error > tol && iter < iter_max )
```

```
{
```

```
  error=0.0;
```

A, Anew resident on host

Copy

```
#pragma acc kernels
```

A, Anew resident on accelerator

These copies
happen every
iteration of the
outer while loop!*

```
  for( int j = 1; j < n-1; j++) {  
    for( int i = 1; i < m-1; i++) {  
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                          A[j-1][i] + A[j+1][i]);  
      error = max(error, abs(Anew[j][i] - A[j][i]));  
    }  
  }
```

Copy

A, Anew resident on accelerator

A, Anew resident on host

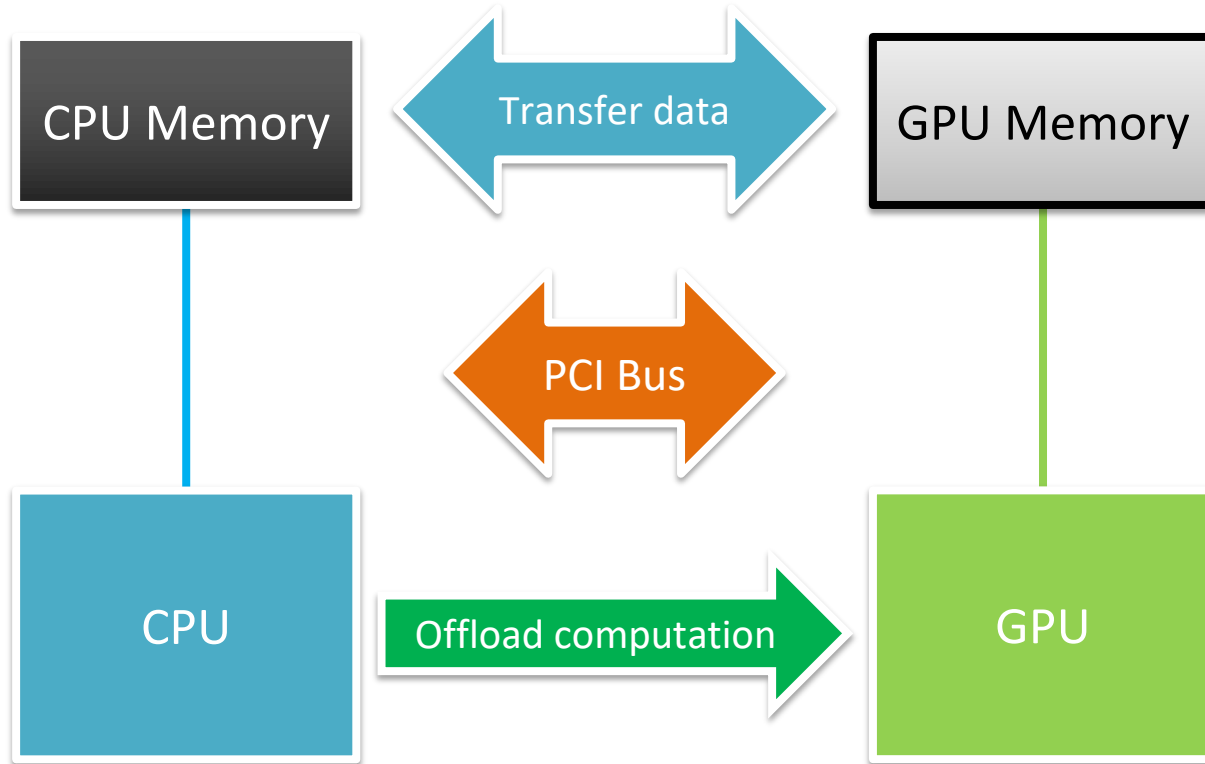
...

```
}
```

OpenACC Fundamentals

Basic Concepts

For efficiency, decouple data movement and compute off-load



OpenACC Fundamentals

Data Construct

- Manage data movement. Data regions may be nested.

```
#pragma acc data[clause ...]  
{ structured block }
```

Clauses

```
if( condition )  
async( expression )
```

Data Clauses	Comment
<code>copy (list)</code>	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
<code>copyin (list)</code>	Allocates memory on GPU and copies data from host to GPU when entering region.
<code>copyout (list)</code>	Allocates memory on GPU and copies data to the host when exiting region.
<code>create (list)</code>	Allocates memory on GPU but does not copy
<code>present (list)</code>	Data is already present on GPU from another containing data region

OpenACC Fundamentals

Example: Second Try - Jacobi Iteration

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator

```
#pragma acc data copy(A) , create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                A[j-1][i] + A[j+1][i]);

            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```

OpenACC Fundamentals

Update Construct

- Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)
 - ✓ Move data from GPU to host, or host to GPU.
 - ✓ Data movement can be conditional, and asynchronous.

```
#pragma update data[clause ...]  
{ structured block }
```

Clauses

```
if(condition) host(list)  
async(expression) device(list)
```

OpenACC Fundamentals

Parallel Construct

```
#pragma acc parallel[clause ...]  
{ structured block }
```

Clauses

```
if( condition )  
async( expression )  
any data clause
```

Parallel Clauses	Comment
<code>num_gangs (expression)</code>	Controls how many parallel gangs are created
<code>num_workers (expression)</code>	Controls how many workers are created in each gang
<code>vector_length (list)</code>	Controls vector length of each worker (SIMD execution)
<code>private(list)</code>	A copy of each variable in list is allocated to each gang
<code>firstprivate (list)</code>	<code>private</code> variables initialized from host
<code>reduction(operator:list)</code>	<code>private</code> variables combined across gangs

OpenACC Fundamentals

Loop Construct

- Detailed control of the parallel execution of the following loop.

```
#pragma acc loop[clause ...]  
    { loop }
```

Loop Clauses	Comment
<code>collapse(n)</code>	Applies directive to the following n nested loops
<code>seq</code>	Executes the loop sequentially on the GPU
<code>private(list)</code>	A copy of each variable in list is allocated to each gang
<code>reduction(operator:list)</code>	private variables combined across gangs

OpenACC Fundamentals

Loop Construct

Loop Clauses inside Parallel Region	Comment
<code>gang</code>	Shares iterations across the gangs of the parallel region
<code>worker</code>	Shares iterations across the workers of the gang
<code>vector</code>	Execute the iterations in SIMD mode

Loop Clauses inside Kernel Region	Comment
<code>gang [(num_gangs)]</code>	Shares iterations across across at most <code>num_gangs</code> gangs
<code>worker [(num_workers)]</code>	Shares iterations across at most <code>num_workers</code> of a single gang
<code>vector [(vector_length)]</code>	Execute the iterations in SIMD mode with maximum <code>vector_length</code>
<code>independent</code>	Specify that the loop iterations are independent

Next Steps

- Get ready for next lecture:
B.4. Shared-memory Parallel Processing
- Get ready for second **hands-on**:
H2. OpenMP Programming (on Cannon)
Check your Cannon account

Questions

Accelerated Computing

