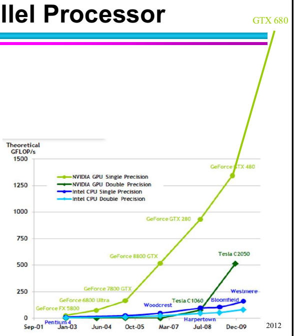
Introduction to NVIDIA CUDA

1

Why Massively Parallel Processor

A quiet revolution and potential build-up (G80 numbers)

- Calculation: 544 GFLOPS vs. 264.96 GFLOPS (FP-64)
- Memory Bandwidth: 153.6GB/s vs. 25.6 GB/s
- Until recently, programmed through graphics API
- GPU in every PC and workstation – massive volume and potential impact



2

Future Apps in Concurrent World

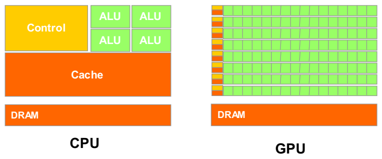
- Exciting applications in future mass computing market
  - Molecular dynamics simulation
  - Video and audio coding and manipulation
  - 3D imaging and visualization
  - Consumer game physics
  - Virtual reality products
- Various granularities of parallelism exist, but…
  - programming model must not hinder parallel implementation
  - data delivery needs careful management
- Introducing domain-specific architecture
  - CUDA for GPGPU

3

What is GPGPU?

- General Purpose computation using GPU in applications (other than 3D graphics)
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD (single-instruction multiple-data) parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

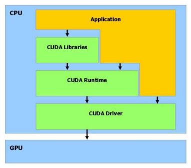
GPU and CPU: The Differences



- GPU
  - More transistors devoted to computation, instead of caching or flow control
  - Suitable for data-intensive computation
    - –High arithmetic/memory operation ratio

CUDA

- “Compute Unified Device Architecture”
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

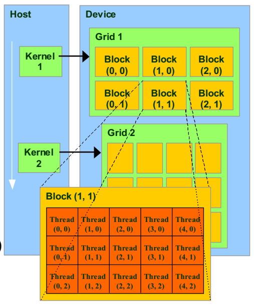


CUDA Programming Model

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
    - Hardware switching between threads (in 1 cycle) on long-latency memory reference
    - Overprovision (1000s of threads) → hide latencies
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

- Kernel executed as a grid of thread blocks
  - All threads share data memory space
- Thread block is a batch of threads, can cooperate with each other by:
  - Synchronizing their execution: For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
  - (Unless thru slow global memory)
- Threads and blocks have IDs



Courtesy: NDVIA

Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];


  __syncthreads()
  ...
  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

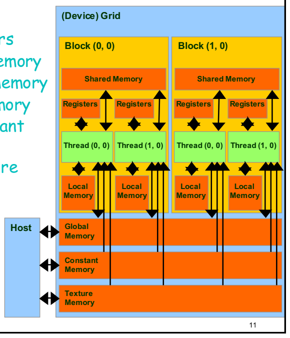
CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void  KernelFunc()`	device	Host
`__host__   float HostFunc()`	host	Host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

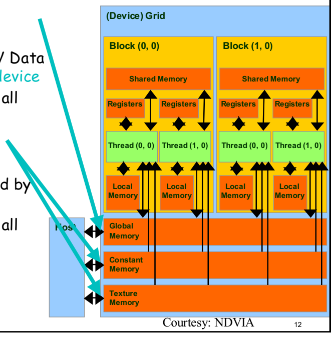
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



Courtesy: NDVIA

Calling Kernel Function – Thread Creation

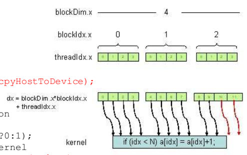
- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous (CUDA 1.0 & later), explicit synch needed for blocking
- Recursion in kernels supported (in 5.0/Kepler+)

Sample Code: Increment Array

```
main() { float *a_h, *a_d; int i, N=10; size_t size = N*sizeof(float);
  a_h = (float *)malloc(size);
  for (i=0; i<N; i++) a_h[i] = (float)i;

  // allocate array on device
  cudaMalloc((void **) &a_d, size);

  // copy data from host to device
  cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);

  // do calculation on device:
  // Part 1 of 2. Compute execution configuration
  int blockSize = 4;
  int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
  // Part 2 of 2. Call incrementArrayOnDevice kernel
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);

  // Retrieve result from device and store in b_h
  cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

  // cleanup
  free(a_h);
  cudaFree(a_d);
}
```

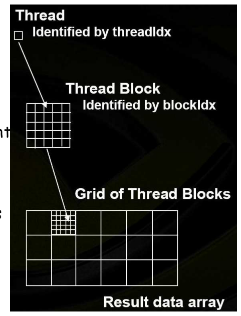


```
__global__ void incrementArrayOnDevice(float *a,
int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx]+1.f;
}
```

Execution model

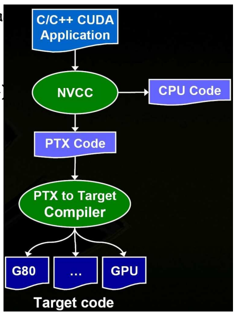
Multiple levels of parallelism

- Thread block
  - Max. 1024 threads/block
  - Communication through shared memory (fast)
  - Thread guaranteed to be resident
  - threadIdx, blockIdx
  - __syncthreads()
    → barrier for this block only!
    avoid RAW/WAR/WAW hazards when ref' shared/global memory
- Grid of thread blocks
  - F<<<nblocks, nthreads>>>(a, b, c)



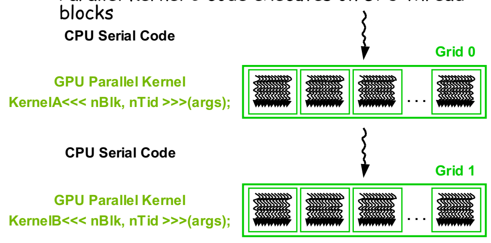
Compiling CUDA

- Call nvcc (driver) -- also C++/Fortra
- LLVM front end (used to be EDG)
  - Separate GPU & CPU code
- LLVM back end (used to be Open64)
  - Generates GPU TPX assembly
- Parallel Threads eXecution (PTX)
  - Virtial machine and ISA
  - Programming model
  - Execution resources and state
- Extensions
  - OpenACC: see ARC web page, like OpenMP but for GPUs
  - OpenCL (not covered here)



Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks



```
KernelA<<< nBlk, nTid >>>(args);
KernelB<<< nBlk, nTid >>>(args);
```

Hardware Implementation: Execution Model

- Each thread block of a grid is split into warps, each gets executed by one multiprocessor (SM)
  - The device processes only one grid at a time
- Each thread block is executed by one multiprocessor
  - So that the shared memory space resides in the on-chip shared memory
- A multiprocessor can execute multiple blocks concurrently
  - Shared memory and registers are partitioned among the threads of all concurrent blocks
  - So, decreasing shared memory usage (per block) and register usage (per thread) increases number of blocks that can run concurrently

Threads, Warps, Blocks

- There are (up to) 32 threads in a Warp
  - Only <32 when there are fewer than 32 **total** threads
- There are (up to) 32 Warps in a Block
- Each Block (and thus, each Warp) executes on a single SM
- GF110 has 16 SMs
- At least 16 Blocks required to "fill" the device
- More is better
  - If resources (registers, thread space, shared memory) allow, more than 1 Block can occupy each SM

More Terminology Review

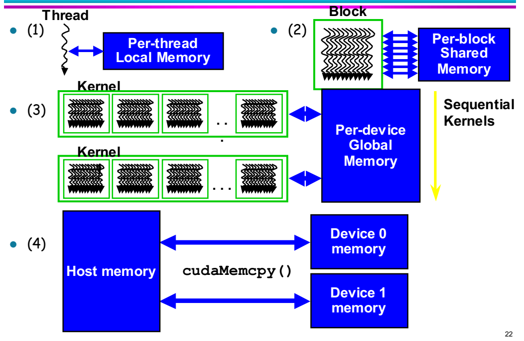
- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

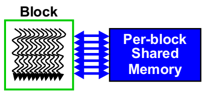
Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

Memory Hierarchy



Using per-block shared memory



- Variables shared across block

```
int *begin, *end;
```

- Scratchpad memory

```
__shared__ int scratch[blocksize];
scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

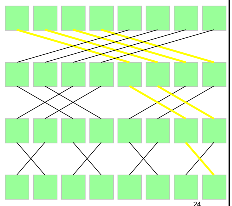
```
scratch[threadIdx.x] = begin[treadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

Example: Parallel Reduction

- Summing up a sequence with 1 thread:

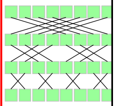
```
int sum = 0;
for(int i=0; i<N; ++i)  sum += x[i];
```

- Parallel reduction builds a summation tree
  - each thread holds 1 element
  - stepwise partial sums
  - N threads need log N steps
  - one possible approach: Butterfly pattern



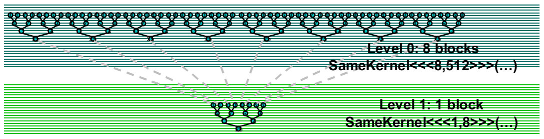
Parallel Reduction for 1 Block

```
// INPUT: Thread i holds value x_i
int i = threadIdx.x;
__shared__ int sum[blocksize];

// One thread per element
sum[i] = x_i; __syncthreads();

for(int bit=blocksize/2; bit>0; bit/=2)
{
  int t=sum[i]+sum[i^bit]; __syncthreads();
  sum[i]=t;                __syncthreads();
}
// OUTPUT: Every thread now holds sum in sum[i]
```



Parallel Reduction Across Blocks

- Code lets B-thread block reduce B-element array
- For larger sequences:
  - reduce each B-element subsequence with 1 block
  - write N/B partial sums to temporary array
  - repeat until done



- P.S. this works for min, max, *, and friends too
  - as written requires associative & commutative function
  - can restructure to work with any associative function

Language Extensions

Built-in Variables

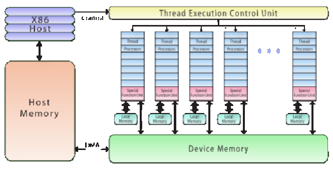
- dim3 gridDim;
  - Dimensions of the grid in blocks (gridDim.z unused)
- dim3 blockDim;
  - Dimensions of the block in threads
- dim3 blockIdx;
  - Block index within the grid
- dim3 threadIdx;
  - Thread index within the block

- Math Functions:
  sin, cos, tan, asin, ...
- Math device functions:
  __sin, ... (faster, less accurate)
- Atomic device functions:
  atomicAdd(), atomicCAS(),...
  - Can implement locks

In Kernel Memory Management

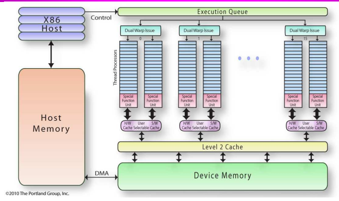
- malloc()
- free()

Tesla Architecture



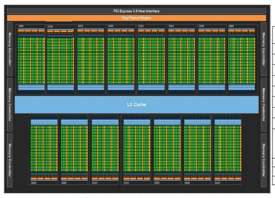
- Used for Technical and Scientific Computing
- L1/L2 Data Cache
  - — Allows for caching of global and local data
  - — Same on-chip memory used for Shared and L1
  - — Configurable at kernel invocation

Fermi Architecture



- L1 cache for each SM
  - — Shared memory/L1: use same memory
  - — Configurable partitions at kernel invocation
    - — 48KB **shared**/16KB **L1** or 16KB **shared**/48KB **L1**
- Unified 768KB L2 Data Cache
  - — Services all load, store, and texture requests

Kepler Architecture



	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

- GK104/K10 early 2012)
- — Configurable shared memory access bank width: 4 / 8 bytes
  - — cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte);...
- GK110/K20 (late 2012)
  - — Dynamic parallelism, HyperQ, more regs/thread & DP throughput

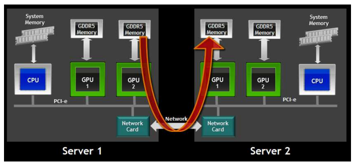
CUDA Toolkit Libraries

NVIDIA GPU-accelerated math libraries:

- cuFFT – Fast Fourier Transforms Library
- cuBLAS – Complete BLAS library
- cuSPARSE – Sparse Matrix library
- cuRAND – Random Number Generation (RNG) Library
- Performance improved since 3.1
- For more info see
  - http://www.nvidia.com/object/gtc2010-presentation-archive.html
- CULA – linear algebra library (commercial add-on)
  - Single precision version free, double costs $s
- Thrust: C++ template lib → STL-like
  - Boost-like saxpy:
    thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), a * _1 + _2);

Libraries & More

- Object linking
  - Plug-ins, libraries
- Dynamic parallelism
  - GPU threads can launch new kernels
- RDMA from GPU(node1) → GPU(node2)



Tools

- Visual Profiler
  - Where is the time spent?
- CUDA-gdb: debugger
- Parallel Nsight + Eclipse
  - Debugger
  - Memory checker
  - Traces (CPU vs. GPU activity)
  - Profiler (memory, instruction throughput, stall)
- Nvidia-smi
  - Turn off ECC
  - Read performance counters

Timing CUDA Kernels

- Real-time Event API

```
cudaEvent_t cstart, cstop;
float cdiff;
cudaEventCreate(&cstart);
cudaEventCreate(&cstop);
cudaEventRecord( cstart, 0 );
kernel<<<x,y,z>>>(a,b,c,);
cudaEventRecord( cstop, 0 );
cudaEventSynchronize( cstop );
cudaEventElapsedTime( &cdiff, cstart, cstop );
Printf("CUDA time is %.3f usec\n", cdiff);
cudaEventDestroy( cstart );
cudaEventDestroy( cstop );
```

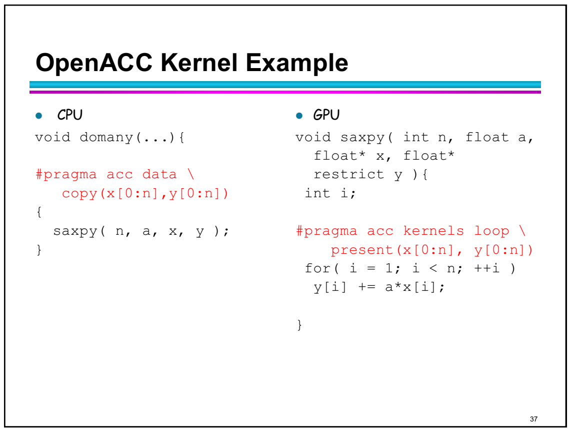
Device Capabilities

- Need to compile for specific capability when needed
  - Flags in Makefile
- Capability levels:
  - 1.0: basic GPU (e.g., 8800 GTX)
  - 1.1: 32-bit atomics in global memory (e.g., GTX 280)
  - 1.2: 64-bit atomics in global+shared memory, warp voting
  - 1.3: double precision floating point
    - e.g., GTX 280/GTX 480, C1060/C1070, C2050/C2070
  - 2.0: caches for global+shared memory
    - e.g., GTX 480, C2050/C2070
  - 3.0: more wraps, threads, blocks, registers…
    - E.g., GTX 680
  - 3.5: Dynamic parallelism, HyperQ
    - E.g., Tesla K20?

OpenACC

- Pragma-based Industry standard, the "OpenMP for GPUs", V1.0
  - #pragma acc [clause]
  - For GPUs but also other accelerat
  - For CUDA but also OpenCL…
- Data movement: sync/async
- Parallelism
- Data layout and caching
- Scheduling
- Mixes w/ MPI, OpenMP
- Works with C, Fortran

OpenACC
DIRECTIVES FOR ACCELERATORS



OpenACC Kernel Example

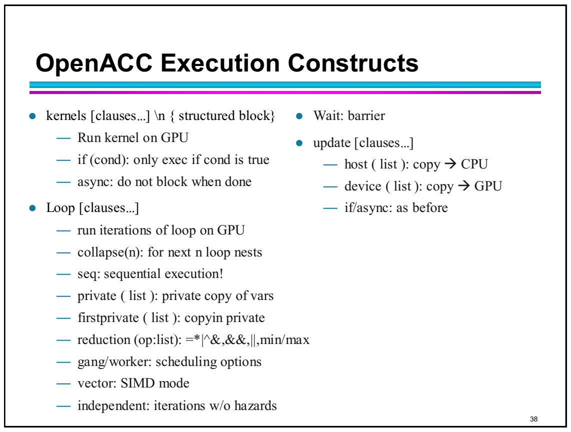
- CPU

```
void domany(...){

#pragma acc data \
   copy(x[0:n],y[0:n])
{
  saxpy( n, a, x, y );
}
```

- GPU

```
void saxpy( int n, float a,
  float* x, float*
  restrict y ){
 int i;

#pragma acc kernels loop \
    present(x[0:n], y[0:n])
 for( i = 1; i < n; ++i )
  y[i] += a*x[i];

}
```

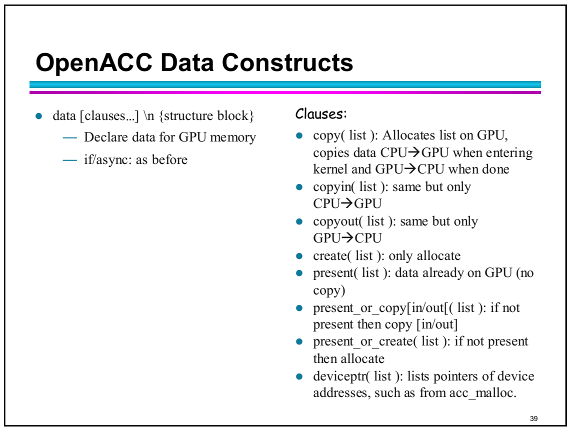
37



OpenACC Execution Constructs

- kernels [clauses...] \n { structured block}
  - Run kernel on GPU
  - if (cond): only exec if cond is true
  - async: do not block when done
- Loop [clauses...]
  - run iterations of loop on GPU
  - collapse(n): for next n loop nests
  - seq: sequential execution!
  - private (list): private copy of vars
  - firstprivate (list): copyin private
  - reduction (op:list): =*|^&,&&,||,min/max
  - gang/worker: scheduling options
  - vector: SIMD mode
  - independent: iterations w/o hazards
- Wait: barrier
- update [clauses...]
  - host (list): copy → CPU
  - device (list): copy → GPU
  - if/async: as before

38



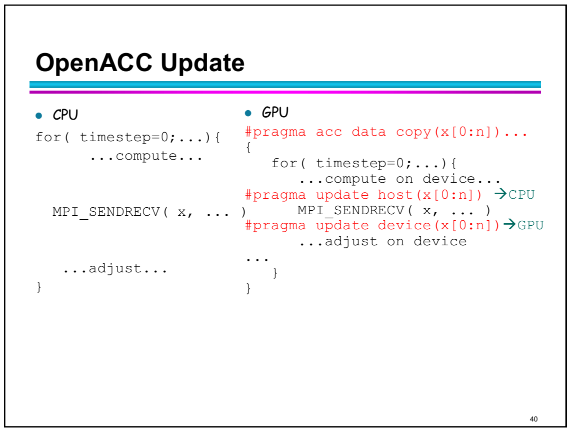
OpenACC Data Constructs

- data [clauses...] \n {structure block}
  - Declare data for GPU memory
  - if/async: as before

Clauses:

- copy(list): Allocates list on GPU, copies data CPU→GPU when entering kernel and GPU→CPU when done
- copyin(list): same but only CPU→GPU
- copyout(list): same but only GPU→CPU
- create(list): only allocate
- present(list): data already on GPU (no copy)
- present_or_copy[in/out[(list): if not present then copy [in/out]
- present_or_create(list): if not present then allocate
- deviceptr(list): lists pointers of device addresses, such as from acc_malloc.

39



OpenACC Update

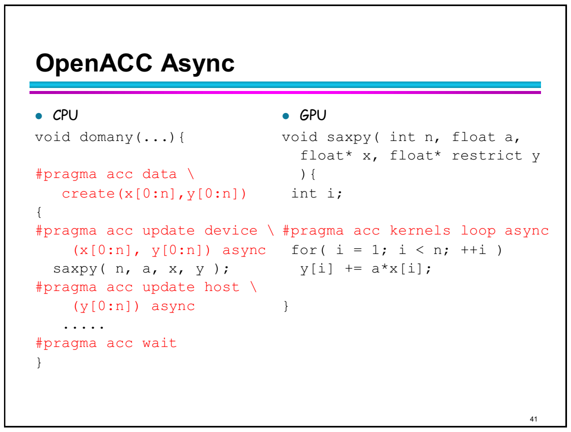
- CPU

```
for( timestep=0;...){
      ...compute...

  MPI_SENDRECV( x, ... )

   ...adjust...
}
```

- GPU

```
#pragma acc data copy(x[0:n])...
{
   for( timestep=0;...){
      ...compute on device...
#pragma update host(x[0:n]) →CPU
      MPI_SENDRECV( x, ... )
#pragma update device(x[0:n])→GPU
      ...adjust on device
...
   }
}
```

40



OpenACC Async

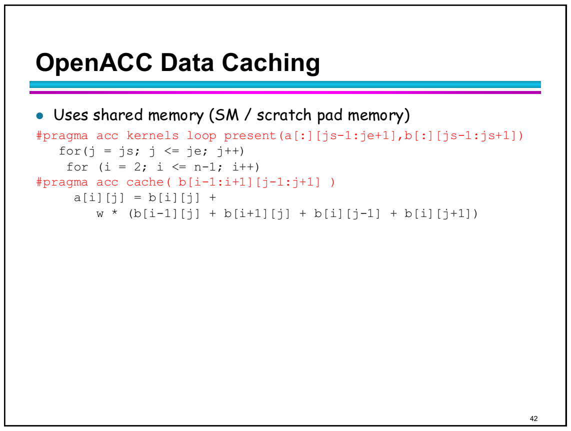
- CPU

```
void domany(...){

#pragma acc data \
   create(x[0:n],y[0:n])
{
#pragma acc update device \
    (x[0:n], y[0:n]) async
  saxpy( n, a, x, y );
#pragma acc update host \
    (y[0:n]) async
   .....
#pragma acc wait
}
```

- GPU

```
void saxpy( int n, float a,
  float* x, float* restrict y
  ){
 int i;
#pragma acc kernels loop async
 for( i = 1; i < n; ++i )
  y[i] += a*x[i];

}
```

41



OpenACC Data Caching

- Uses shared memory (SM / scratch pad memory)

```
#pragma acc kernels loop present(a[:][js-1:je+1],b[:][js-1:js+1])
   for(j = js; j <= je; j++)
    for (i = 2; i <= n-1; i++)
#pragma acc cache( b[i-1:i+1][j-1:j+1] )
     a[i][j] = b[i][j] +
        w * (b[i-1][j] + b[i+1][j] + b[i][j-1] + b[i][j+1])
```

42

OpenACC Parallel / Loop (for)

- GPU Parallel

```
#pragma acc parallel \
    copy(x[0:n],y[0:n])
{
  saxpy( n, a, x, y );
}
```

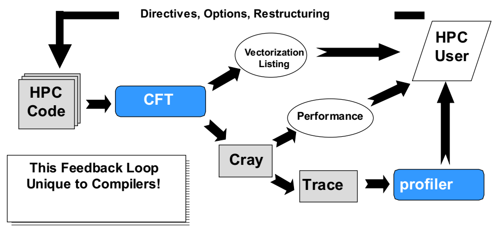
- GPU Loop

```
void saxpy( int n, float a,
  float* x, float*
  restrict y ){
 int i;

#pragma acc loop
 for( i = 1; i < n; ++i )
  y[i] += a*x[i];

}
```

OpenACC Runtime Constructs

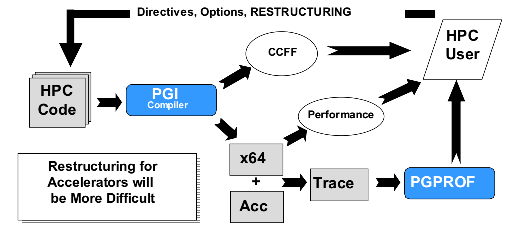
- #include "openacc.h"
- acc_malloc(size_t)
- acc_free(void*)
- acc_async_test(expression)
- acc_async_test_all()
- acc_async_wait(expression)
- acc_async_wait_all()

Cray OpenACC


Directives, Options, Restructuring
HPC User
Vectorization Listing
HPC Code
CFT
Performance
This Feedback Loop Unique to Compilers!
Cray
Trace
profiler

We can use this same methodology to enable effective migration of applications to Multi-core and Accelerators

PGI OpenACC


Directives, Options, RESTRUCTURING
CCFF
HPC User
HPC Code
PGI Compiler
Performance
x64
+
Acc
Trace
PGPROF
Restructuring for Accelerators will be More Difficult