OpenACC Advanced

Directive-based GPGPU Tuning

Sandra Wienke, M.Sc.
wienke@rz.rwth-aachen.de
Center for Computing and Communication
RWTH Aachen University

PPCES, March 2013

Agenda

- **Kernel optimizations**
  - → Loop schedule
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

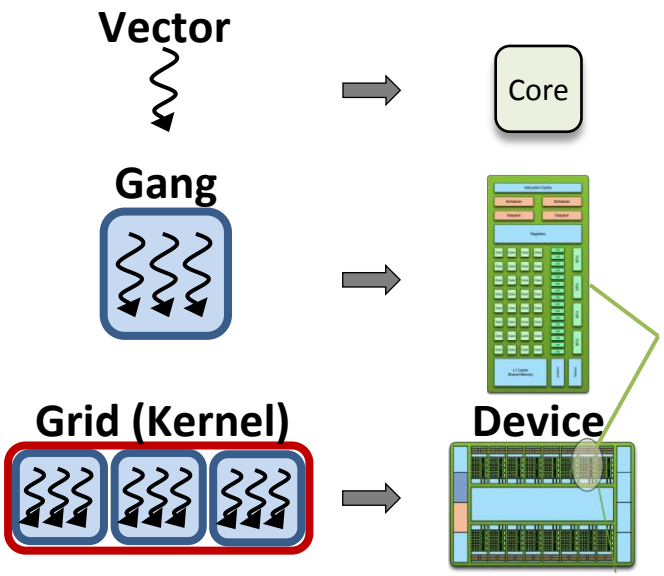
Mapping OpenACC on CUDA model

- **Threads execute as groups of 32 threads (warps)**
  - → Threads in warp share same program counter

➔ **SIMT architecture**

- **Possible mapping to CUDA terminology (GPUs)** *compiler dependent*
  - → gang = block
  - → worker = warp
  - → vector = threads
    - → Within block (if omitting worker)
    - → Within warp (if specifying worker)

- **Execution Model**



Loop schedule *(in examples)*

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{

#pragma acc kernels copy(y[0:n]) present(x[0:n])
#pragma acc loop gang vector(256)
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

#pragma acc kernels copy(y[0:n]) present(x
#pragma acc loop gang(64) vector
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

Without loop schedule

```
33, Loop is parallelizable
    Accelerator kernel generated
    33, #pragma acc loop gang,
    vector(128)
39, Loop is parallelizable
    Accelerator kernel generated
    39, #pragma acc loop gang,
    vector(128)
```

With loop schedule

```
33, Loop is parallelizable
    Accelerator kernel generated
    33, #pragma acc loop gang,
    vector(256)
39, Loop is parallelizable
    Accelerator kernel generated
    39, #pragma acc loop
    gang(64), vector(128)
```

Sometimes the compiler feedback is buggy. Check once with ACC_NOTIFY.

Loop schedule *(in examples)*

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{

#pragma acc parallel copy(y[0:n]) present(x[0:n]) vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

#pragma acc parallel copy(y[0:n]) present(x[0:n]) num_gangs(64)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

`vector_length`: Optional: Specifies number of threads in thread block.
`num_gangs`: Optional. Specifies number of thread blocks in grid.

Loop schedule *(in examples)*

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
       // do something
  }
```

Distributes loop to **n** threads on GPU.
→256 threads per thread block
→usually `ceil(n/256)` blocks in grid

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
     for (int j = 0; j < m; ++j){
          // do something
  }  }
```

Distributes outer loop to **n** threads on GPU.
Each thread executes inner loop sequentially.
→256 threads per thread block
→usually `ceil(n/256)` blocks in grid

```
#pragma acc parallel vector_length(256) num_gangs(16)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
       // do something
  }
```

Distributes loop to threads on GPU (see above). If 16*256 < n, each thread gets multiple elements.
→256 threads per thread block
→16 blocks in grid

Loop schedule *(in examples)*

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang
  for (int i = 0; i < n; ++i){
#pragma acc loop vector
      for (int j = 0; j < m; ++j){
            // do something
  }  }
```

Distributes outer loop to GPU multiprocessors (block-wise). Distributes inner loop to threads within thread blocks.

→256 threads per thread block

→usually **n** blocks in grid

```
#pragma acc kernels
#pragma acc loop gang(100) vector(8)
  for (int i = 0; i < n; ++i){
#pragma acc loop gang(200) vector(32)
      for (int j = 0; j < m; ++j){
            // do something
  }  }
```

With nested loops, specification of multidimensional blocks and grids possible: use same resource for outer and inner loop.

→100 blocks in Y-direction (rows); 200 blocks in X-direction (columns)

→8 threads in Y-dimension of one block; 21 threads in X-dimension of one block

→Total: 8*32=256 threads per thread block; 100*200=20,000 blocks in grid

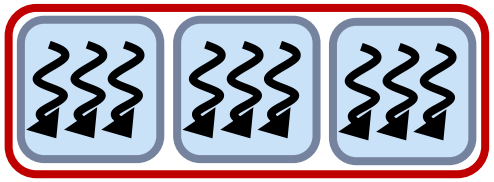
Multidimensional portioning not yet possible with **parallel** construct.
→ See proposed **tile** clause in v2.0

Agenda

- **Kernel optimizations**
  - → Loop schedule
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

Launch configuration

- **Kernel executes grid of blocks/gangs of threads/vectors**



  → Launch configuration

- **How many threads/blocks to launch?**
- **Hardware operation**
  - → Instructions are issued in order
  - → Thread execution stalls when one of the operands isn't ready
  - → Latency is hidden by switching threads
    - → GMEM latency: 400-800 cycles
    - → Arithmetic latency: 18-22 cycles

→ **Need enough threads to hide latency**

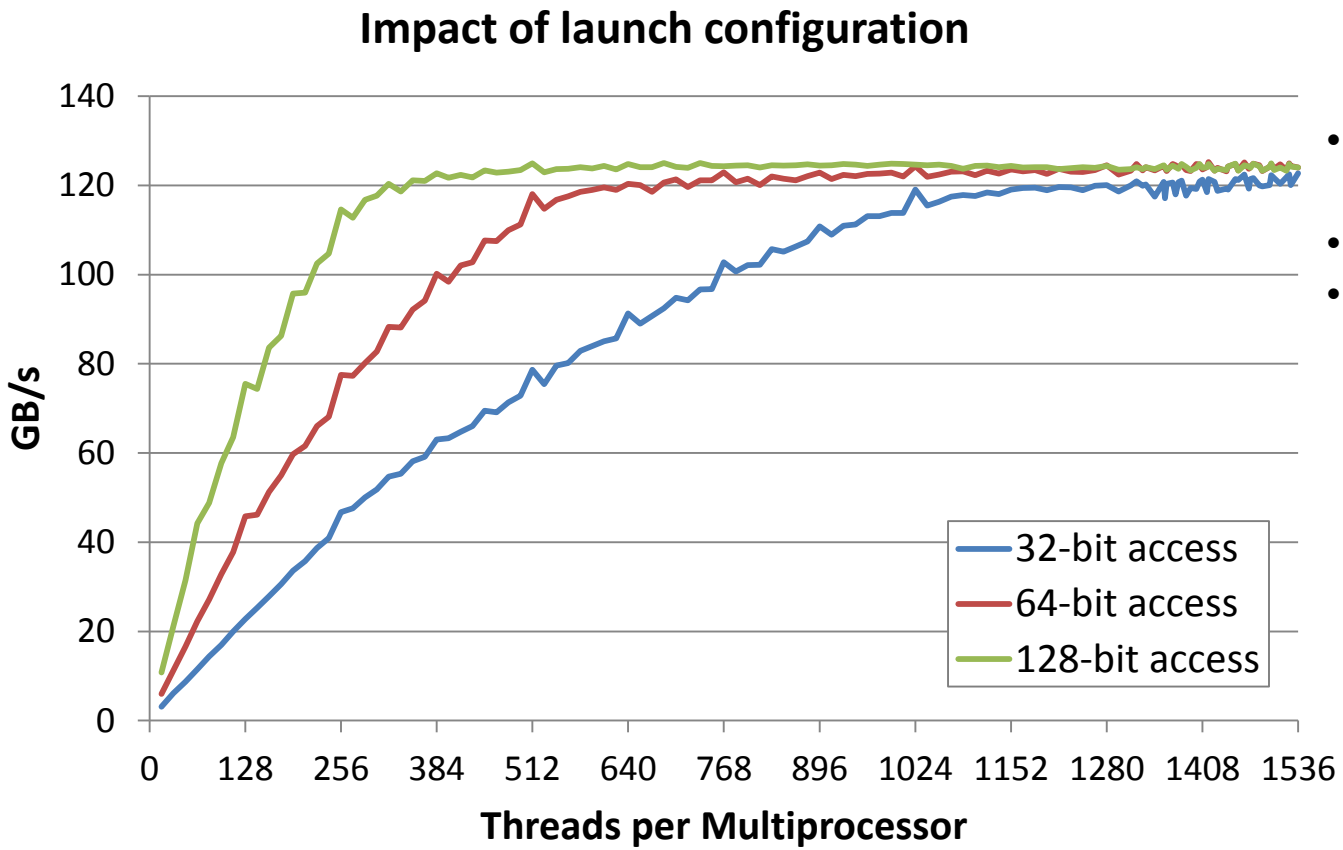
Launch configuration

- **Hiding arithmetic latency**
  - → Need ~18 warps (576 threads) per Fermi MP
  - → Or, independent instructions from the same warp

- **Maximizing global memory throughput**
  - → Gmem throughput depends on the access pattern, and word size
  - → Need enough memory transactions in flight to saturate the bus
    - → Independent loads and stores from the same thread
    - → Loads and stores from different threads
    - → Larger word sizes can also help

Impact of launch configuration & word size

- **Maximizing global memory throughput**
  - → Example program: increment an array of ~67M elements



- NVIDIA Tesla C2050 (Fermi)
- ECC off
- Bandwidth: 144 GB/s

How many threads/ blocks to launch?

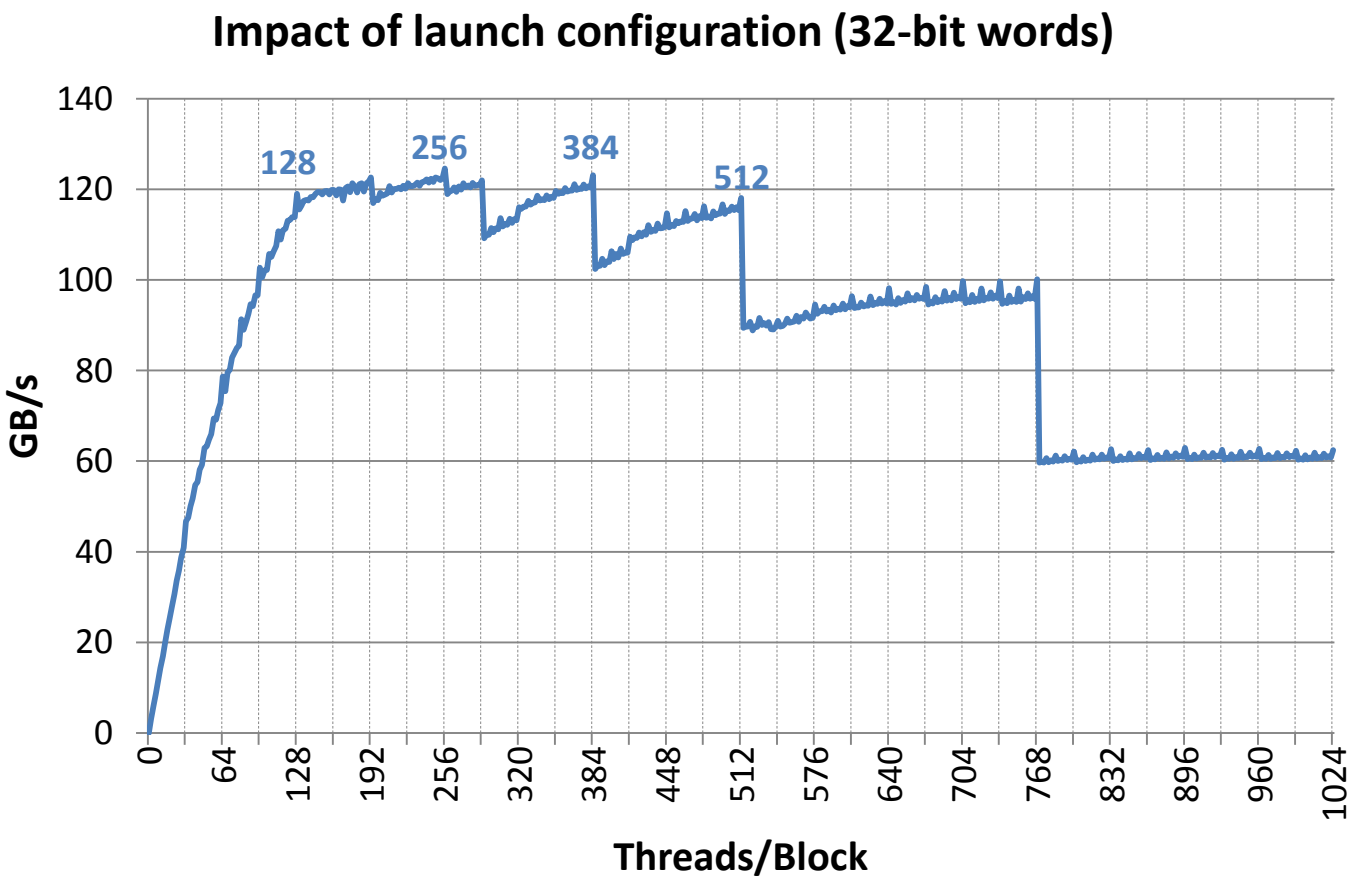
- **Threads per block (=vectors per gang): Multiple of warp size (32)**
  - → Not used threads are marked as inactive
  - → Starting point: 256-512 threads/block
- **Blocks (gangs) per grid heuristics**
  - → **`#blocks > #MPs`**
    - → MPs have at least one block to execute
  - → **`#blocks / #MPs > 2`**
    - → MP can concurrently execute up to 8 blocks
    - → Blocks that aren't waiting in barrier keep hardware busy
    - → Subject to resource availability (registers, smem)
  - → **`#blocks > 50`** to scale to future devices
  - → Most obvious: **`#blocks * #threads = #problem-size`**
    - → Multiple elements per thread may amortize setup costs of simple kernels:

      `#blocks * #threads < #problem-size`

→ **Launch MANY threads to keep GPU busy!**

Impact of launch configuration

- **Maximizing global memory throughput**
  - → Example program: increment an array of ~67M elements



- NVIDIA Tesla C2050 (Fermi)
- ECC off
- Bandwidth: 144 GB/s

Agenda

- **Kernel optimizations**
  - → Loop schedule
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

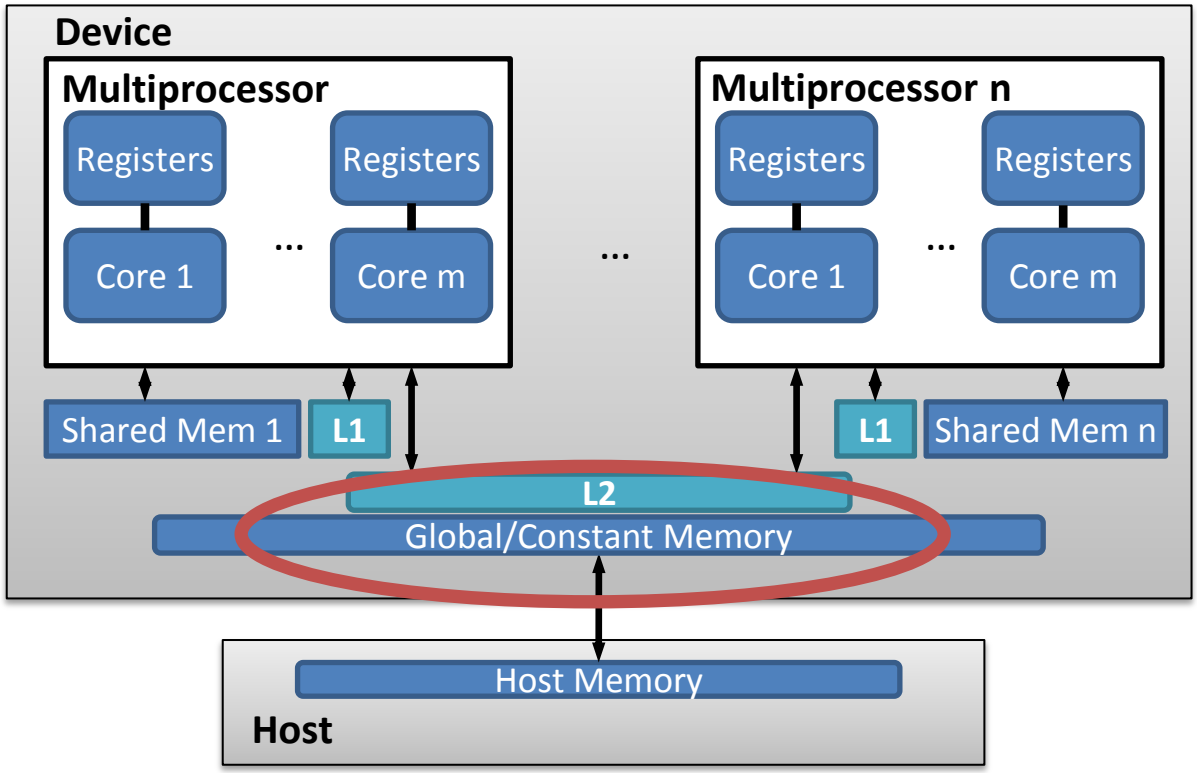
Background information

- **NVIDIA‘s compute capability (cc)**
  - → Describes architecture and features of GPU
  - → E.g. number of registers
  - → E.g. double precision computations (only since cc 1.3)
  - → E.g. management of memory accesses (per 16/32 threads)

- **Examples**
  - → GT200, Tesla C1060: cc 1.3
  - → Fermi C2050, Quadro 6000: cc 2.0
  - → Kepler K20: cc 3.5

Recap memory hierarchy

- **Local memory/ Registers**
- **Shared memory/ L1**
  - → Very low latency (~100x than gmem)
  - → Bandwidth (aggregate): 1+ TB/s
- **L2**
- **Global memory**
  - → High latency (400-800 cycles)
  - → Bandwidth: 144 GB/s



gmem access

- **Stores:** Invalidate L1, write-back for L2
- **Loads:**

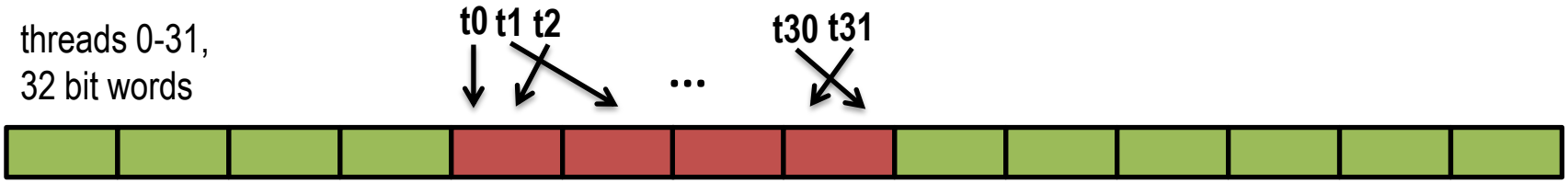
	Caching	Non-caching
Enabling by	default	Experimental compiler-option PGI: `-Mx,180,8`
Attempt to hit:	L1 → L2 → gmem	L2 → gmem (no L1: invalidate line if it's already in L1)
Load granularity	128-byte line	32-byte line

→ Threads in a warp provide memory addresses

→ Determine which lines/segments are needed

→ Request the needed lines/segments

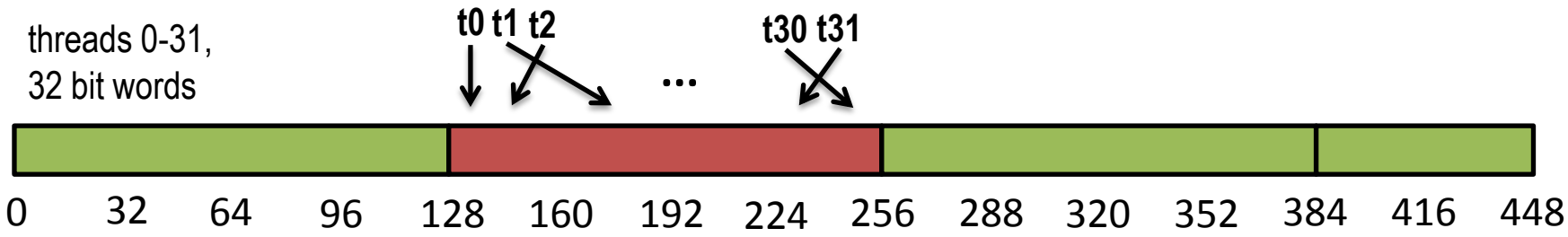
Gmem access per warp (32 threads)



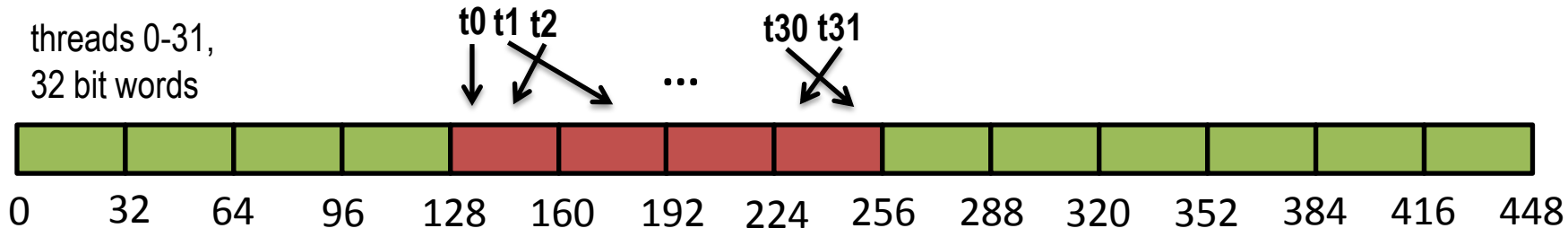
gmem throughput

Example 1: Warp requests 32 aligned, permuted 4-byte words → 128 bytes needed

- **Caching load**
  - → Addresses fall within 1 cache-line:
    128 bytes move across the bus on a miss → **100 %** bus utilization



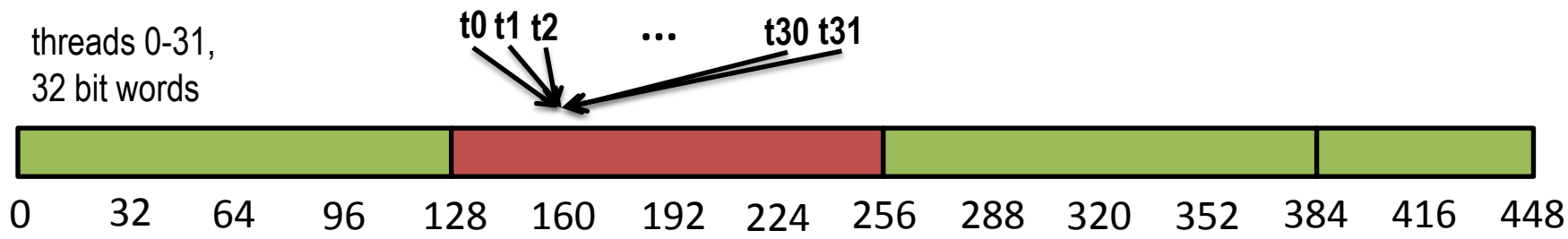
- **Non-caching load**
  - → Addresses fall within 4 segments
    128 bytes move across the bus on a miss → **100 %** bus utilization



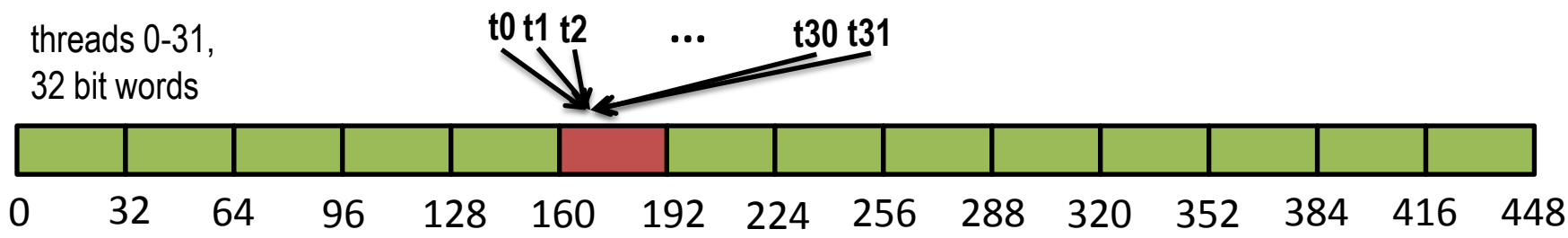
gmem throughput

Example 2 (*broadcast*): All threads in a warp request the same 4-byte word
→ 4 bytes needed

- **Caching load**
  - → Addresses fall within 1 cache-line:
    128 bytes move across the bus on a miss → **3.125 %** bus utilization

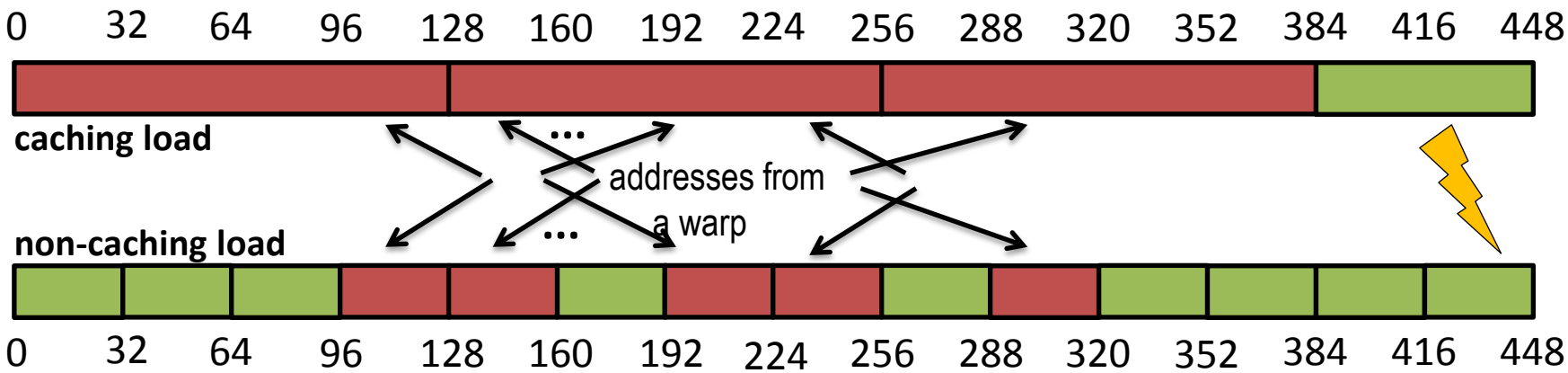


- **Non-caching load**
  - → Addresses fall within 1 segment
    32 bytes move across the bus on a miss → **12.5 %** bus utilization



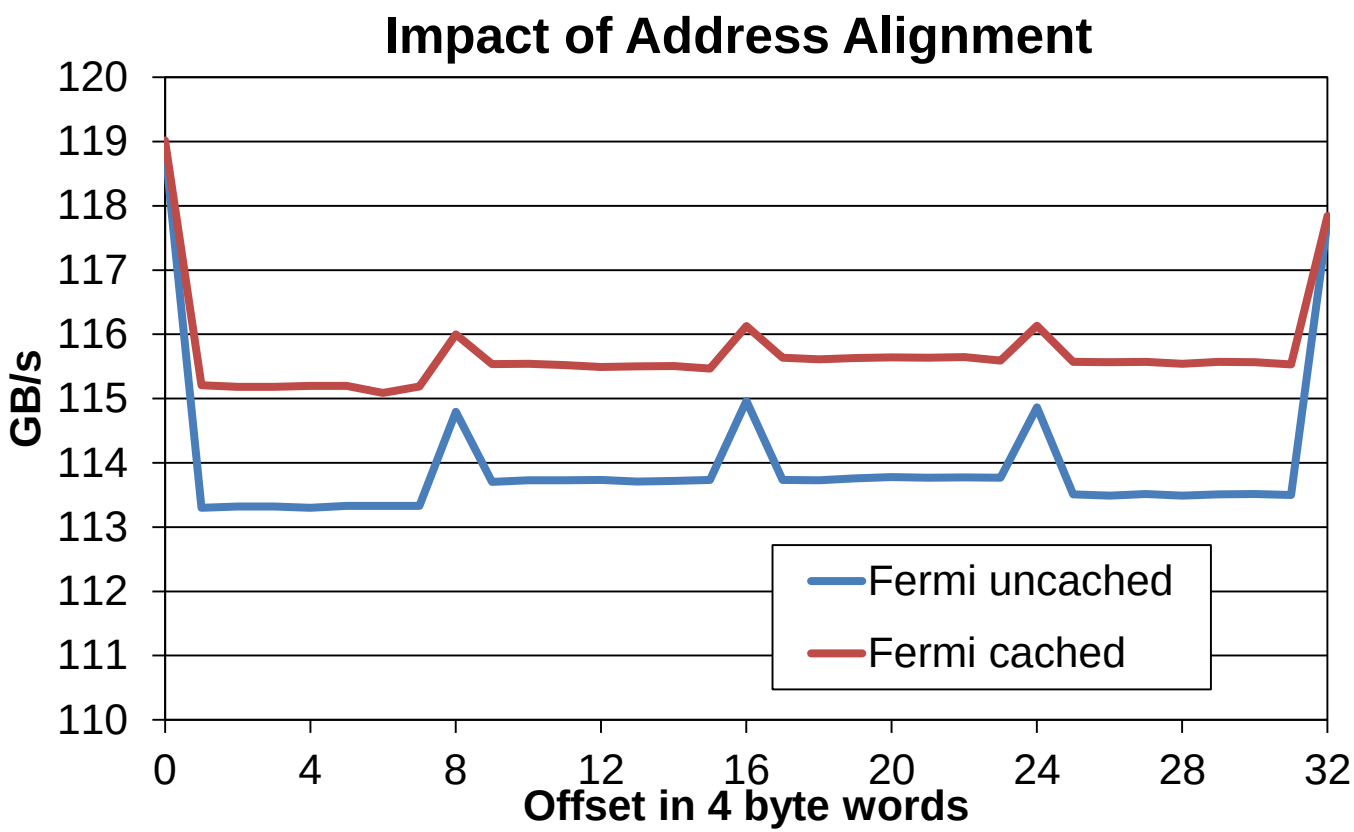
gmem throughput – coalescing

- **Range of accesses**



- **Address alignment**

Impact of address alignment



Example program:

- Copy ~67 MB of floats
- NVIDIA Tesla C2050 (Fermi)
- ECC off
- 256 threads/block

→ **Misaligned accesses can drop memory throughput**

→ Fermi L1 can help for misaligned loads

→ ECC can reduce misaligned store throughput

Data layout matters (improve coalescing)

- **Example: AoS vs. SoA**

Array of Structures (AoS)

```
struct myStruct_t {
  float a;
  float b;
  int c;
}
myStruct_t myData[];
```

Structure of Arrays (SoA)

```
struct {
  float a[];
  float b[];
  int c[];
}
```

```
#pragma acc kernels
for (int i=0; i<n; i++) {
  … myData[i].a / myData.a[i] …
}
```

Address	0	4	8	12	16	20	24	28	32
AoS	A[0]	B[0]	C[0]	A[1]	B[1]	C[1]	A[2]	B[2]	C[2]
SoA	A[0]	A[1]	A[2]	B[0]	B[1]	B[2]	C[0]	C[1]	C[2]

Conclusion gmem throughput

- **Strive for perfect coalescing**
  - → Warp should access within contiguous region
  - → Align starting address (may require padding)

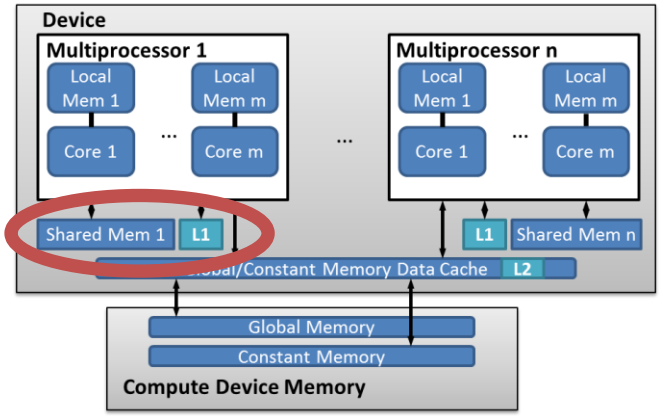
- **Have enough concurrent accesses to saturate the bus**
  - → Process several elements per thread
  - → Launch enough threads to cover access latency

Agenda

- **Kernel optimizations**
  - → Control flow
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

Shared memory

- **Smem per MP (10s of KB)**
  - → Inter-thread communication within a block
  - → Need synchronization to avoid RAW / WAR / WAW hazards
- **Low-latency, high-throughput memory**
  - → Cache data in smem to reduce gmem accesses




```
#pragma acc kernels copyout(b[0:N][0:N]) copyin(a[0:N][0:N])
#pragma acc loop gang vector
  for (int i = 1; i < N-1; ++i){
#pragma acc loop gang vector
   for (int j = 1; j < N-1; ++j){
#pragma acc cache(a[i-1:i+1][j-1:j+1])
      b[i][j] = (a[i][j-1] + a[i][j+1] +
                 a[i-1][j] + a[i+1][j]) / 4;
  }    }
```

```
Accelerator kernel generated
65, #pragma acc loop gang, vector(2)
    Cached references to size
    [(y+2)x(x+2)] block of 'a'
67, #pragma acc loop gang, vector(128)
```

Directives: Caching

- **Cache construct**
  - → Prioritizes data for placement in the highest level of data cache on GPU

C/C++

```
#pragma acc cache(list)
```

Fortran

```
!$acc cache(list)
```

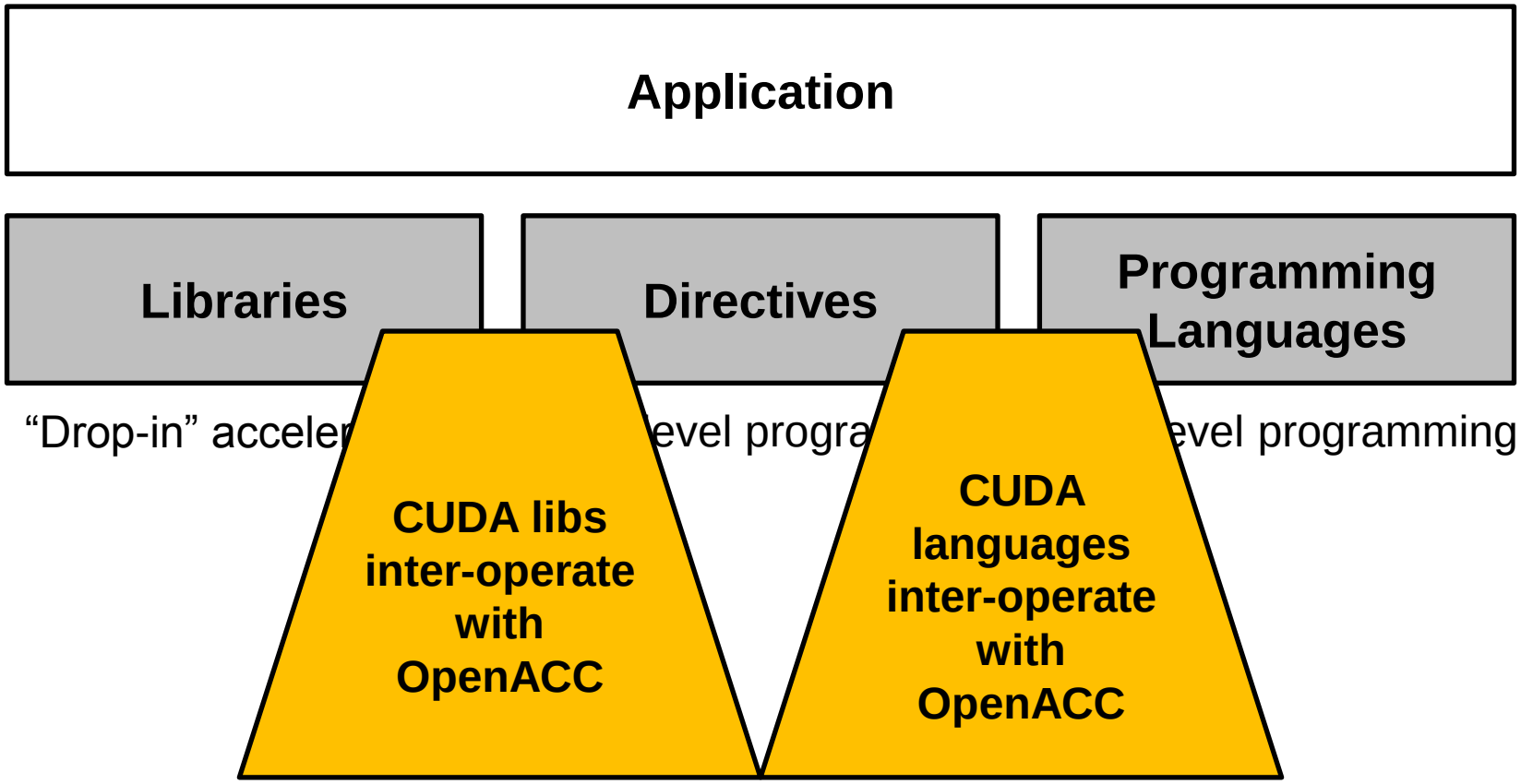
Sometimes the PGI compiler ignores the `cache` construct.
→ See compiler feedback

  - → Use at the beginning of the loop or in front of the loop

Agenda

- **Kernel optimizations**
  - → Control flow
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

Interoperability



Libraries

- **NVIDIA**
  - → cuBLAS — Dense linear algebra (complete BLAS)
  - → cuSPARSE — Sparse linear algebra
  - → cuFFT — Discrete Fourier transforms
  - → cuRAND — Random number generation
  - → NPP — Performance Primitives for Image & Video Processing
  - → Thrust — STL/Boost style template lib (e.g. scan, sort, reduce, transform)
  - → math.h — Basics, exponentials, trigonometry,.. (e.g. sin, ceil, round)
- **Third party**
  - → CULA — Dense/sparse linear algebra (subset of LAPACK)
  - → MAGMA — Dense linear algebra (subset of BLAS, LAPACK)
  - → IMSL — Fortran numerical library utilizes cuBLAS
  - → NAG — Numeric libraries (e.g. RNGs)
  - → libJacket — Math, signal processing, image processing, statistics
  - → GPU VSIPL — Vector signal image processing
- **Open Source**
  - → cuDPP — Data parallel primitives (e.g. scan, sort, reduction)
  - → CUSP — Sparse linear algebra, graph computations
  - → OpenCurrent — Partial differential equations
- **...**

Interoperability

- **CUDA and OpenACC both operate on device arrays**
  - → Interoperability with CUDA libraries
  - → Interoperability with custom CUDA C/C++/Fortran code

→ **Make device address available on host or in OpenACC code**

```
#pragma acc data copy(x[0:n])
{
#pragma acc kernels loop
  for (int i=0; i<n; i++) {
    // work on x[i]
  }
#pragma acc host_data use_device(x)
  {
    // use device pointer x:
    // - for library
    // - for custom CUDA code
} }
```

```
cudaMalloc(&x, sizeof(float)*n);
// use device pointer x:
// - for library
// - for custom CUDA code


#pragma acc data deviceptr(x)
{
#pragma acc kernels loop
  for (int i=0; i<n; i++) {
    // work on x[i]
} }
```

Directives: deviceptr & host_data

- **Deviceptr clause**
  - → Declares that pointers in *list* refer to device pointers that need not be allocated/ moved between host and device

C/C++
```
#pragma acc parallel|kernels|data deviceptr(list)
```

Fortran
```
!$acc data deviceptr(list)
```

- **Host_data construct**
  - → Makes the address of device data in *list* available on the host
  - → Vars in *list* must be present in device memory

C/C++
```
#pragma acc host_data use_device(list)
```

Fortran
```
!$acc host_data use_device(list)

!$acc end host_data
```

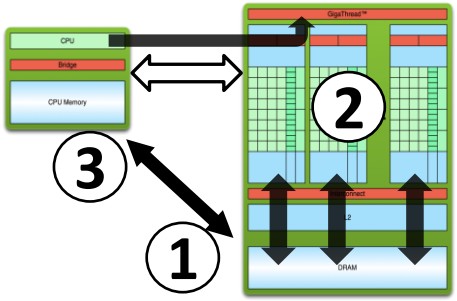
only valid clause

Agenda

- **Kernel optimizations**
  - → Loop schedule
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

Minimal data transfer

- **Data copies between host + device**
  - → PCI Express x16 Gen2: 8 GB/s
  - → GPU memory bandwidth (Fermi): 144 GB/s
  - → Sync. kernel executions/ memory copies block CPU thread



- ➔ **Minimize CPU/GPU idling + maximize PCIe throughput**
  - → Copy as little data as possible
  - → Copy as rarely as possible
  - → Batch small transfers into one larger one
  - → Async operations

Agenda

- **Kernel optimizations**
  - → Loop schedule
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

Asynchronous operations

- **Definition**
  - → Synchronous: Control does not return until accelerator action is complete
  - → Asynchronous: Control returns immediately
    - → Allows heterogeneous computing (CPU + GPU)
- **Asynchronous operations with `async` clause**
  - → Kernel execution: `kernels`, `parallel`
  - → Data movement: `update`, PGI only: `data` construct

```
#pragma acc kernels async
for (int i=0; i<n; i++) {…}

// do work on host
```

Directives: async clause

- **Async clause**
  - → Executes **`parallel`** | **`kernels`** | **`update`** operation asynchronously while host process continues with code

C/C++

```
#pragma acc parallel|kernels|update async[(scalar-int-expr)]
```

Fortran

```
!$acc parallel|kernels|update async[(scalar-int-expr)]
```

  - → Integer argument (optional) can be seen as CUDA stream number
  - → Integer argument can be used in a wait directive
  - → Async activities with same argument: executed in order
  - → Async activities with diff. argument: executed in any order relative to each other

Streams

- **Streams = CUDA concept**

- **Stream = sequence of operations that execute in issue-order on GPU**
  - → Same int-expr in async clause: one stream
  - → Different streams/ int-expr: any order relative to each other

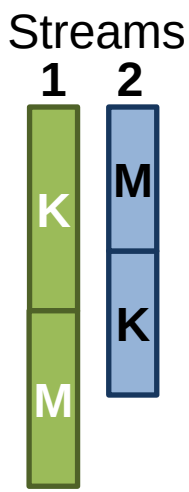
- **Tips for debugging - disable async (PGI only)**
  - → Use int-expr “-1” in async clause (PGI 13.x)
  - → Set ACC_SYNCHRONOUS=1

Wait on asynchronous operations

- **Synchronize async operations → wait directive**
  → Wait for completion of an asynchronous activity (all or certain stream)

```
#pragma acc kernels loop async
for(int i=0; i<N/2; i++) {/* do work on device: x[0:N/2] */}
// do work on host: x[N/2:N]
#pragma acc update host(x[0:N/2]) async
#pragma acc wait
```

```
#pragma acc kernels loop async(1)
for(int i=0; i<N; i++) {/* do work on device: x[0:N] */}
#pragma acc update device(y[0:M]) async(2)
#pragma acc kernels loop async(2)
for(int i=0; i<M; i++) {/* do work on device: y[0:M] */}
#prama acc update host(x[0:N]) async(1)
// do work on host: z[0:L]
#pragma acc wait(1)
// do work on host: x[0:N]
#pragma acc wait(2)
```



Directives: wait

- **Wait directive**
  - → Host thread waits for completion of an asynchronous activity
  - → If integer expression specified, waits for all async activities with the same value
  - → Otherwise, host waits until all async activities have completed

C/C++

```
#pragma acc wait[(scalar-int-expr)]
```

Fortran

```
!$acc wait[(scalar-int-expr)]
```

- **Runtime routines**
  - → **`int acc_async_test(int)`**: tests for completion of all associated async activities; returns nonzero value/.true. if all have completed
  - → **`int acc_asnyc_test_all()`**: tests for completion of all async activities
  - → **`int acc_async_wait(int)`**: waits for completion of all associated async activites; routine will not return until the latest async activity has completed
  - → **`int acc_async_wait_all()`**: waits for completion of all asnyc activities

Agenda

- **Kernel optimizations**
  - → Loop schedule
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

Multiple GPUs

- **Several GPUs within one compute node**
  - → Can use within one program (or use MPI)
  - → E.g. assign one device to each CPU thread/ process

```
#include <openacc.h>

acc_set_device_num(0, acc_device_nvidia);
#pragma acc kernels async

acc_set_device_num(1, acc_device_nvidia);
#pragma acc kernels async
```

Multiple GPUs – OpenMP + MPI

OpenMP

```
#include <openacc.h>
#include <omp.h>

#pragma omp parallel num_threads(12)
{
  int numdev = acc_get_num_devices(acc_device_nvidia);
  int devID = omp_get_thread_num() % numdev;
  acc_set_device_num(devID, acc_device_nvidia);
}
```

MPI

```
#include <openacc.h>
#include <mpi.h>

int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
int numdev = acc_get_num_devices(acc_device_nvidia);
int devID = myrank % numdev;
acc_set_device_num(devID, acc_device_nvidia);
```

Agenda

- **Kernel optimizations**
  - → Loop schedule
  - → Launch configuration
  - → Global memory throughput
  - → Shared memory
  - → Interoperability with CUDA Libs
- **GPU-CPU interaction**
  - → Minimal data transfer
  - → Asynchronous operations
  - → Multiple GPUs
- **Summary**

Summary

- **Try different loop schedules**
- **Kernel launch configuration**
  - → Launch enough threads per MP to hide latency
  - → Launch enough thread blocks to load the GPU
- **Maximize global memory throughput**
  - → GPU has lots of bandwidth, use it effectively
  - → Coalescing, alignment, (non-)caching loads
- **Use shared memory when applicable (over 1 TB/s bandwidth)**
- **Use library/ low-level routines if maximum performance needed**
- **Minimize CPU/GPU idling**
  - → Asynchronous operations
  - → Use all resources (multiple GPUs if applicable)
- **Use analysis/ profiling when optimizing**

Outlook OpenACC 2.0

- **`default(none)` clause for parallel/ kernels**
  - → No implicit data movement by compiler; requires explicit definition by user
- **Unstructured data lifetimes / new data API routines**
  - → `#pragma acc enter data/ exit data`: move data without regions
- **Call support**
  - → `#pragma acc routine [clause]`: supports calls from the accelerator
- **`async` clause on `wait` directive**
  - → Gives programmers a mechanism to resolve dependencies between multiple async handles without requiring the host thread to wait
- **Nested parallelism (`parallel`, `kernels`)**
  - → May give a better performance; to compose program modules
- **`tile` clause for (tightly nested) loops**
  - → Allows multidimensional partitioning and/or cache benefits

German Heterogeneous Computing Group (GHCG)

- **Usergroup Meeting 23.5.-24.5.2013 (2 half days)**
- **Workshop on programming accelerators 22.5.-23.5.2013 (2 half days)**