Lecture 21: Manycore GPU Architectures and Programming, Part 3

-- Streaming, Library and Tuning

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

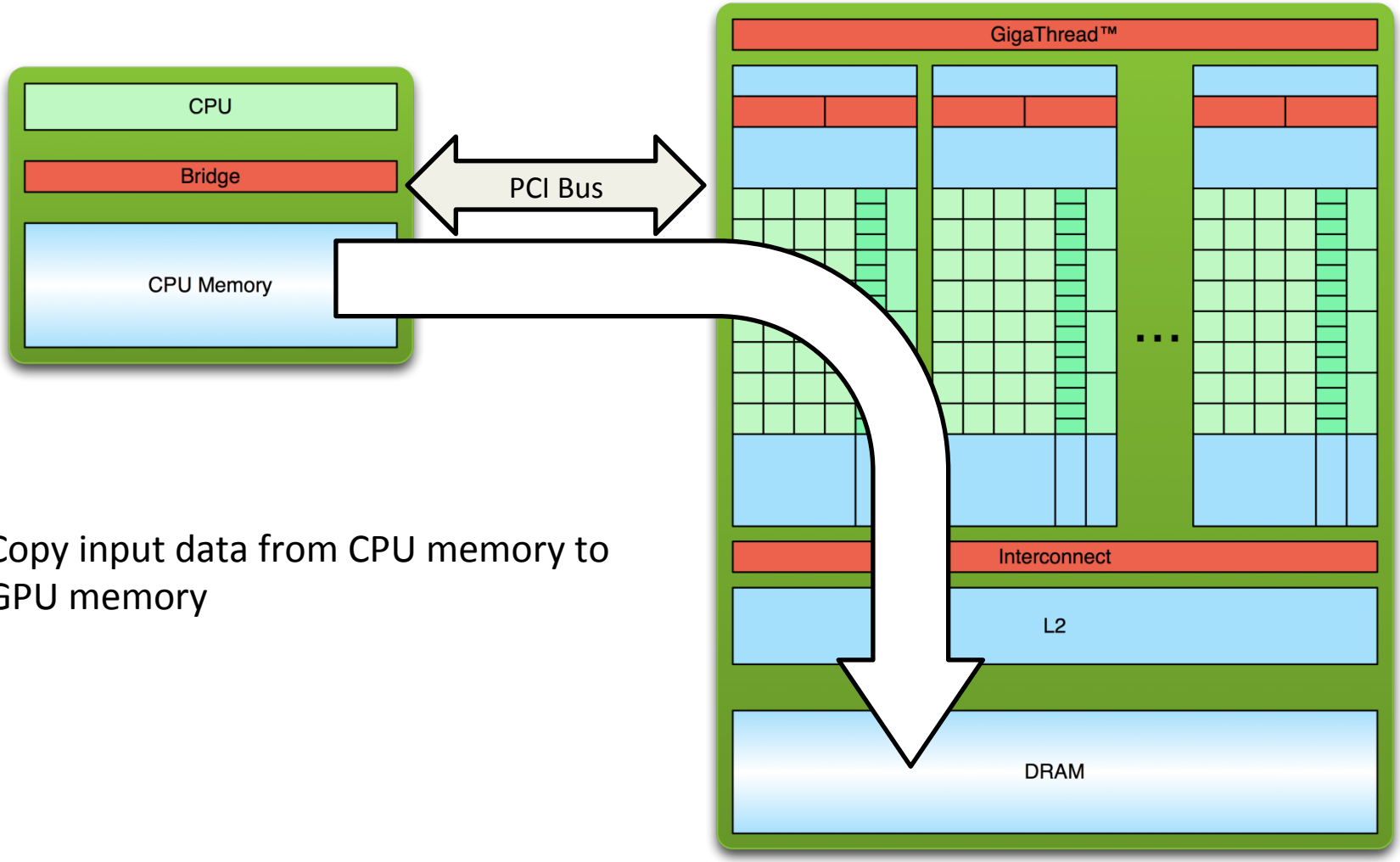
Manycore GPU Architectures and Programming: Outline

- Introduction
  - GPU architectures, GPGPUs, and CUDA
- GPU Execution model
- CUDA Programming model
- Working with Memory in CUDA
  - Global memory, shared and constant memory

☛ **Streams and concurrency**

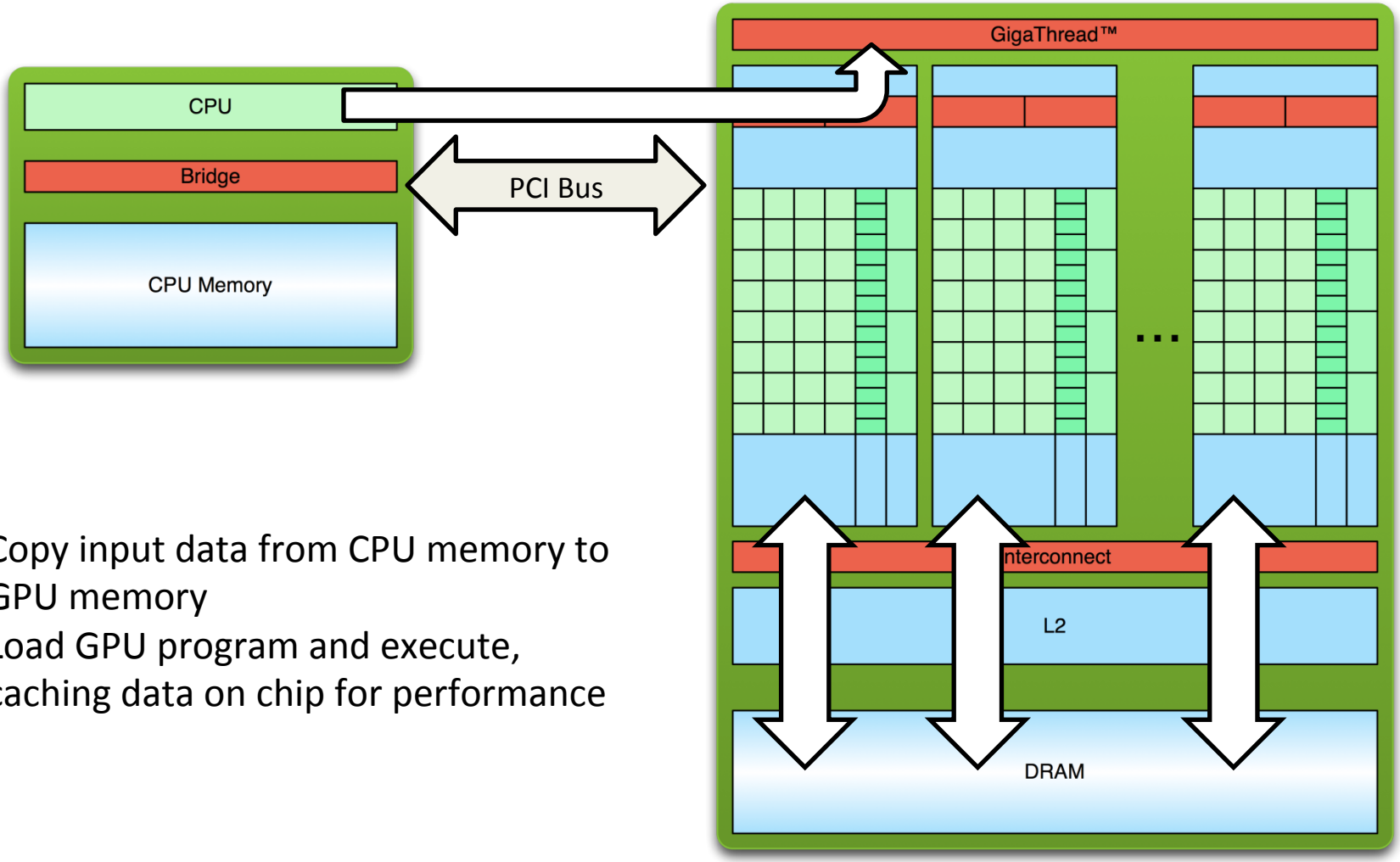
- CUDA ~~instruction intrinsic~~ and library
- Performance, profiling, debugging, and error handling
- Directive-based high-level programming model
  - OpenACC and OpenMP

Offloading Processing Flow



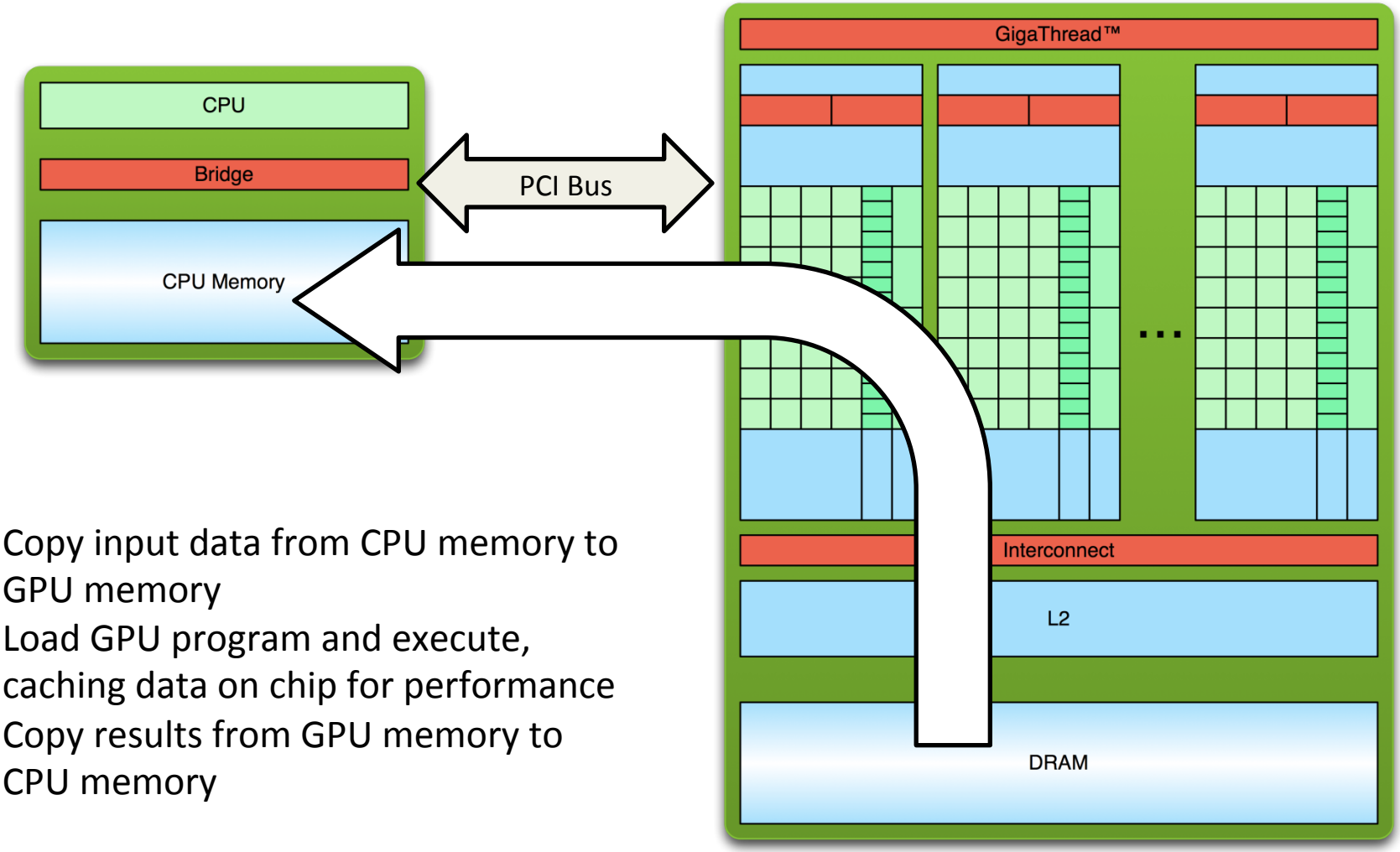
1. Copy input data from CPU memory to GPU memory

Offloading Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

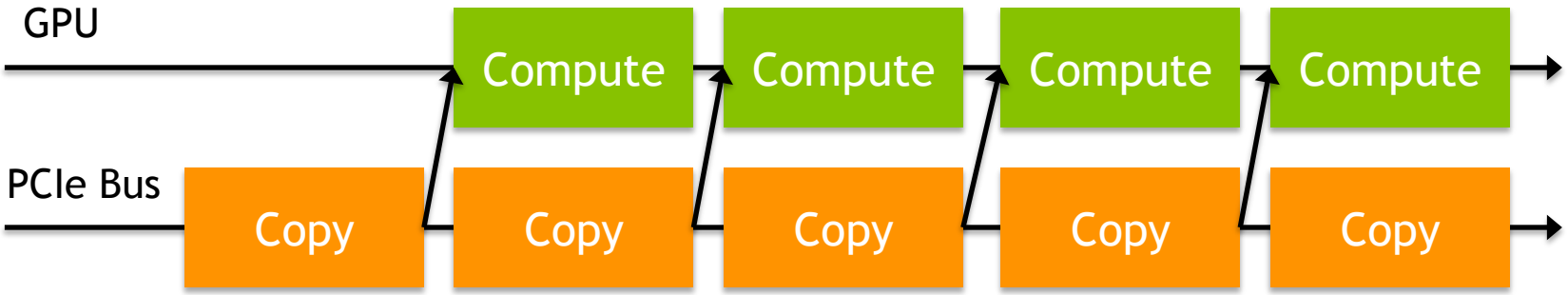
Offloading Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Overlapping Communication and Computation

- Three sequential steps for a single kernel execution
- Multiple kernels
  - Asynchrony is a first-class citizen of most GPU programming frameworks
  - Computation-communication overlap is a common technique in GPU programming



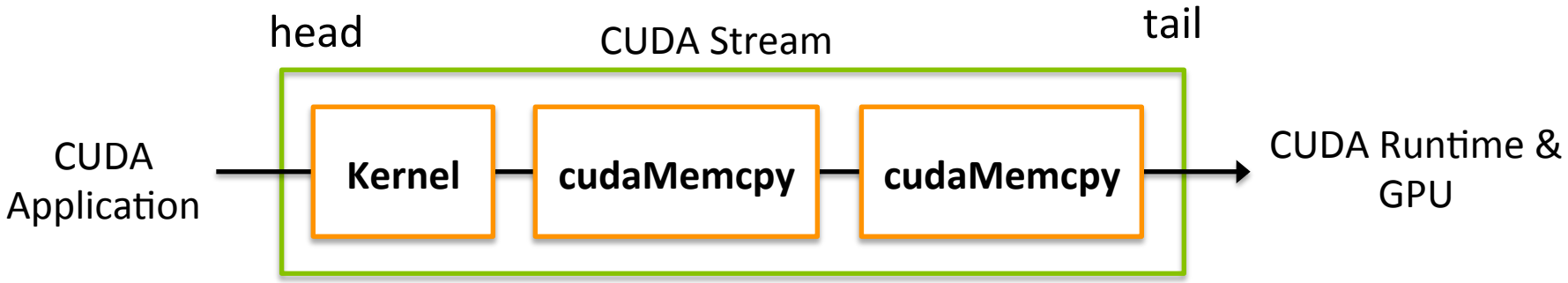
Abstract Concurrency

- Different kinds of action overlap are possible in CUDA?
  1. Overlapped host computation and device computation
  2. Overlapped host computation and host-device data transfer
  3. Overlapped host-device data transfer and device computation
  4. Concurrent device computation

- CUDA Streams to achieve each of these types of overlap

CUDA Streams

- CUDA Streams: a FIFO queue of CUDA actions to be performed
  - Placing a new action at the head of a stream is **asynchronous**
  - Executing actions from the tail as CUDA resources allow
  - Every action (kernel launch, `cudaMemcpy`, etc) runs in an implicit or explicit stream



CUDA Streams

- Two types of streams in a CUDA program
  - The **implicitly** declared stream (**NULL stream**)
  - **Explicitly** declared streams (**non-NULL streams**)

- Up until now, all code has been using the NULL stream by default

```
cudaMemcpy(...);
kernel<<<...>>>(...);
cudaMemcpy(...);
```

- Non-NULL streams require manual allocation and management by the CUDA programmer

CUDA Streams

- To create a CUDA stream:

```
cudaError_t cudaStreamCreate(cudaStream_t *stream);
```

- To destroy a CUDA stream:

```
cudaError_t cudaStreamDestroy(cudaStream_t stream);
```

- To wait for all actions in a CUDA stream to finish:

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
```

- To check if all actions in a CUDA stream have finished:

```
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

CUDA Streams

- **`cudaMemcpyAsync`**: Asynchronous memcpy

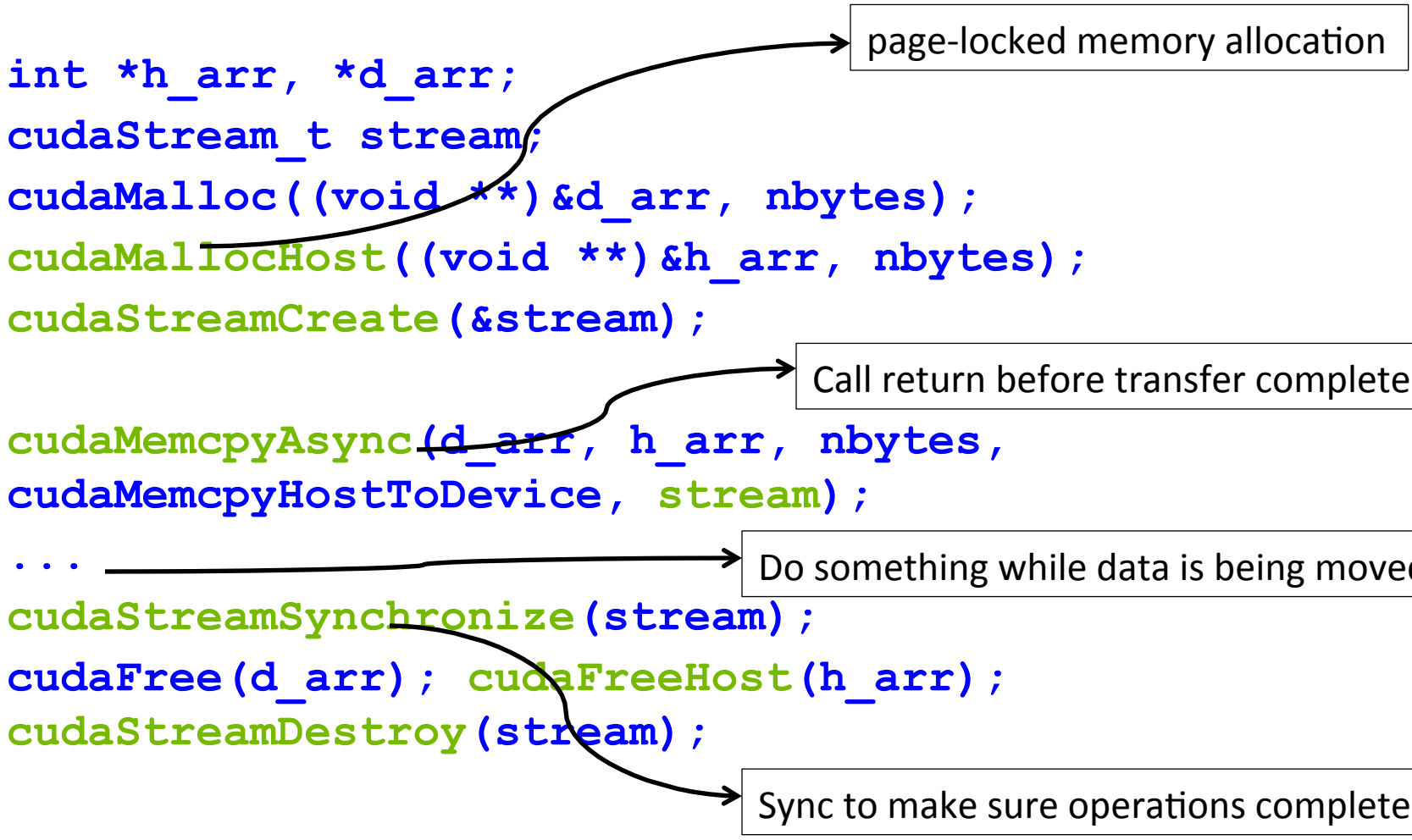
```
cudaError_t cudaMemcpyAsync(void *dst, const void *src,
size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0);
```

- **`cudaMemcpyAsync`** does the same as `cudaMemcpy`, but may return before the transfer is actually complete

- Pinned host memory is a requirement for `cudaMemcpyAsync`
  - Memory that is resident in physical memory pages, and cannot be swapped out, also referred as page-locked
    - Recall malloc normally reserve virtual address space first and then actually physical pages are allocated

CUDA Streams

- Performing a `cudaMemcpyAsync`:



CUDA Streams

- Associate kernel launches with a non-NULL stream
  - Note that kernels are always asynchronous

```
kernel<<<nblocks, threads_per_block,
smem_size, stream>>>(...);
```

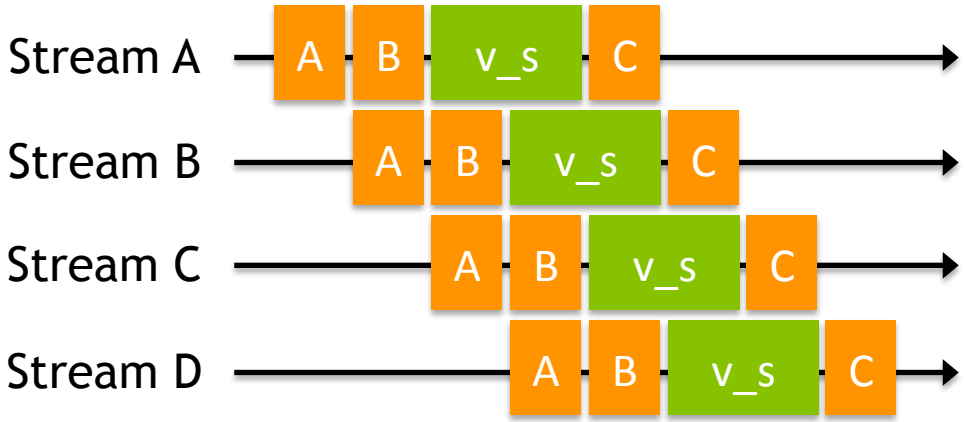
- The effects of `cudaMemcpyAsync` and kernel launching
  - Operations are put in the stream queue for execution
  - Actually operations may not happen yet

- Host-side timer to time those operations
  - Not the actual time of the operations

CUDA Streams

- Vector sum example, A + B = C

- Partition the vectors and use CUDA streams to overlap copy and compute



CUDA Streams

- How can this be implemented in code?

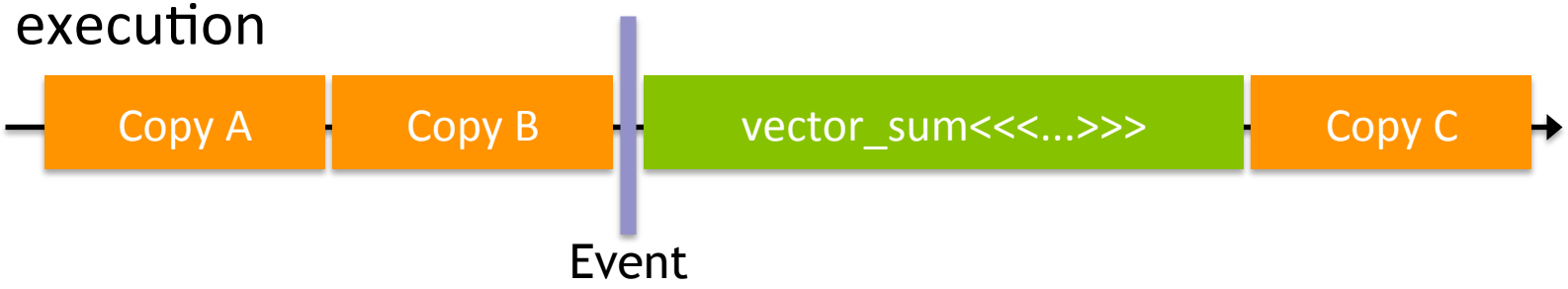
```
for (int i = 0; i < nstreams; i++) {
  int offset = i * eles_per_stream;
  cudaMemcpyAsync(&d_A[offset], &h_A[offset], eles_per_stream *
     sizeof(int), cudaMemcpyHostToDevice, streams[i]);
  cudaMemcpyAsync(&d_B[offset], &h_B[offset], eles_per_stream *
     sizeof(int), cudaMemcpyHostToDevice, streams[i]);
   ……
  vector_sum<<<..., streams[i]>>>(d_A + offset,
     d_B + offset, d_C + offset);
  cudaMemcpyAsync(&h_C[offset], &d_C[offset], eles_per_stream *
     sizeof(int), cudaMemcpyDeviceToHost, streams[i]);
}

for (int i = 0; i < nstreams; i++)
  cudaStreamSynchronize(streams[i]);
```

CUDA Events

- Timing asynchronous operations
  - Host-side timer: only measure the time for the call, not the actual time for the data movement or kernel execution

- Events to streams, which mark specific points in stream execution



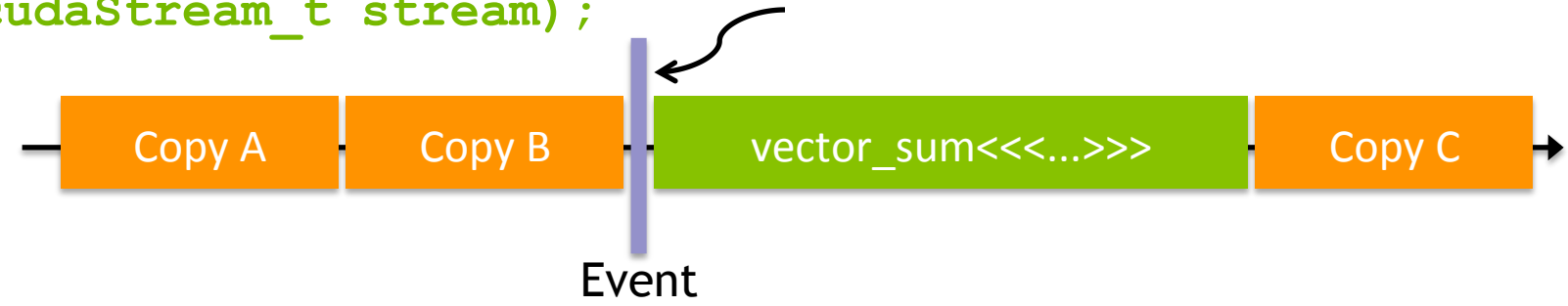
- Events are manually created and destroyed:

```
cudaError_t cudaEventCreate(cudaEvent_t *event);
cudaError_t cudaEventDestroy(cudaEvent_t *event);
```

CUDA Events

- To add an event to a CUDA stream:

```
cudaError_t cudaEventRecord(cudaEvent_t event,
cudaStream_t stream);
```



  - `Event` marks the point-in-time after all preceding actions in `stream` complete, and before any actions added after `cudaEventRecord` run

- Host to wait for some CUDA actions to finish

```
cudaError_t cudaEventSynchronize(cudaEvent_t event);
```

  - Wait for all the operations before this events to complete, but not those after

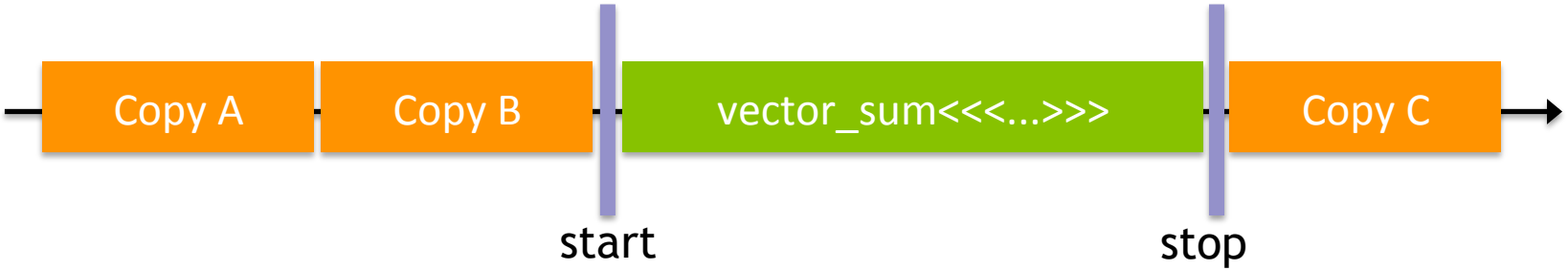
CUDA Events

- Check if an event has been reached without waiting for it:

```
cudaError_t cudaEventQuery(cudaEvent_t event);
```

- Get the elapsed milliseconds between two events:

```
cudaError_t cudaEventElapsedTime(float *ms, cudaEvent_t start, cudaEvent_t stop);
```



CUDA Events

- In codes:

```
float time;
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);

cudaEventRecord(start);
kernel<<<grid, block>>>(arguments);
cudaEventRecord(stop);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Implicit and Explicit Synchronization

- Two types of host-device synchronization:
  - **Implicit synchronization** causes the host to wait on the GPU, but as a side effect of other CUDA actions
  - **Explicit synchronization** causes the host to wait on the GPU because the programmer has asked for that behavior

Implicit and Explicit Synchronization

- Five CUDA operations that include implicit synchronization:
  1. A pinned host memory allocation (`cudaMallocHost`, `cudaHostAlloc`)
  2. A device memory allocation (`cudaMalloc`)
  3. A device memset (`cudaMemset`)
  4. A memory copy between two addresses on the same device (`cudaMemcpy(..., cudaMemcpyDeviceToDevice)`)
  5. A modification to the L1/shared memory configuration (`cudaThreadSetCacheConfig`, `cudaDeviceSetCacheConfig`)

Implicit and Explicit Synchronization

- Four ways to explicitly synchronize in CUDA:
  1. Synchronize on a device

     ```
     cudaError_t cudaDeviceSynchronize();
     ```

  2. Synchronize on a stream

     ```
     cudaError_t cudaStreamSynchronize();
     ```

  3. Synchronize on an event

     ```
     cudaError_t cudaEventSynchronize();
     ```

  4. Synchronize across streams using an event

     ```
     cudaError_t cudaStreamWaitEvent(cudaStream_t
     stream, cudaEvent_t event);
     ```

Implicit and Explicit Synchronization

- **`cudaStreamWaitEvent`** adds inter-stream dependencies
  - Causes the specified `stream` to wait on the specified `event` before executing any further actions
  - `event` does not need to be an event recorded in `stream`

```
cudaEventRecord(event, stream1);
...
cudaStreamWaitEvent(stream2, event);
...
```

  - No actions added to stream2 after the call to `cudaStreamWaitEvent` will execute until event is satisfied

Suggested Readings

1. Chapter 6 in *Professional CUDA C Programming*
2. Justin Luitjens. *CUDA Streams: Best Practices and Common Pitfalls*. GTC 2014. http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
3. Steve Rennich. *CUDA C/C++ Streams and Concurrency.* 2011. http://on-demand.gputechconf .com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

Manycore GPU Architectures and Programming: Outline

- Introduction
  - GPU architectures, GPGPUs, and CUDA
- GPU Execution model
- CUDA Programming model
- Working with Memory in CUDA
  - Global memory, shared and constant memory
- Streams and concurrency
- ☛ **CUDA ~~instruction intrinsic~~ and library**
- Performance, profiling, debugging, and error handling
- Directive-based high-level programming model
  - OpenACC and OpenMP

CUDA Libraries

- CUDA Libraries offer pre-packaged and expertly-optimized functions that implement commonly useful operations.
  - Vector addition, matrix vector, matrix matrix, FFT, etc

CUDA Libraries

- What are the advantages of CUDA Libraries?
  - Support a wide range of application domains
  - Highly usable, high-level APIs that are familiar to domain experts
  - Tuned by CUDA experts to perform well across platforms and datasets
  - Often offer the quickest route for porting, simply swap out API calls
  - Low maintenance, developer of the library takes on responsibility of bug fixes and feature requests

CUDA Libraries

LIBRARY NAME	DOMAIN
NVIDIA cuFFT	Fast Fourier Transforms
NVIDIA cuBLAS	Linear Algebra (BLAS Library)
CULA Tools	Linear Algebra
MAGMA	Next-gen Linear Algebra
IMSL Fortran Numerical Library	Mathematics and Statistics
NVIDIA cuSPARSE	Sparse Linear Algebra
NVIDIA CUSP	Sparse Linear Algebra and Graph Computations
AccelerEyes ArrayFire	Mathematics, Signal and Image Processing, and Statistics
NVIDIA cuRAND	Random Number Generation
NVIDIA NPP	Image and Signal Processing
NVIDIA CUDA Math Library	Mathematics
Thrust	Parallel Algorithms and Data Structures
HiPLAR	Linear Algebra in R
Geometry Performance Primitives	Computational Geometry
Paralution	Sparse Iterative Methods
AmgX	Core Solvers

Workflow to Use CUDA Library

1. Create a library-specific handle that manages contextual information useful for the library's operation.
   - Many CUDA Libraries have the concept of a handle which stores opaque library-specific information on the host which many library functions access
   - Programmer's responsibility to manage this handle
   - For example: `cublasHandle_t`, `cufftHandle`, `cusparseHandle_t`, `curandGenerator_t`

2. Allocate device memory for inputs and outputs to the library function.
   - Use `cudaMalloc` as usual

Common Library Workflow

3. If inputs are not already in a library-supported format, convert them to be accessible by the library.
   - Many CUDA Libraries only accept data in a specific format
   - For example: column-major vs. row-major arrays

4. Populate the pre-allocated device memory with inputs in a supported format.
   - In many cases, this step simply implies a `cudaMemcpy` or one of its variants to make the data accessible on the GPU
   - Some libraries provide custom transfer functions, for example: `cublasSetVector` optimizes strided copies for the CUBLAS library

Common Library Workflow

5. Configure the library computation to be executed.
   - In some libraries, this is a no-op
   - Others require additional metadata to execute library computation correctly
   - In some cases this configuration takes the form of extra parameters passed to library functions, others set fields in the library handle

6. Execute a library call that offloads the desired computation to the GPU.
   - No GPU-specific knowledge required

Common Library Workflow

7. Retrieve the results of that computation from device memory, possibly in a library-determined format.
   - Again, this may be as simple as a `cudaMemcpy` or require a library-specific function

8. If necessary, convert the retrieved data to the application's native format.
   - If a conversion to a library-specific format was necessary, this step ensures the application can now use the calculated data
   - In general, it is best to keep the application format and library format the same, reducing overhead from repeated conversions

Common Library Workflow

9. Release CUDA resources.
   - Includes the usual CUDA cleanup (`cudaFree`, `cudaStreamDestroy`, etc) plus any library-specific cleanup

10. Continue with the remainder of the application.

Common Library Workflow

- Not all libraries follow this workflow, and not all libraries require every step in this workflow
  - In fact, for many libraries many steps are skipped
  - Keeping this workflow in mind will help give you context on what the library might be doing behind the scenes and where you are in the process

- Next, we'll take a look at two commonly useful libraries
  - Try to keep the common workflow in mind while we work with them

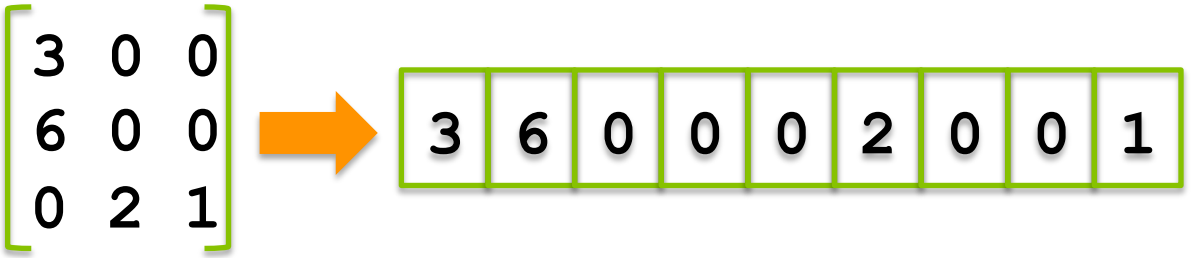
cuBLAS

- cuBLAS is a port of a popular linear algebra library, BLAS

- cuBLAS (like BLAS) splits its subroutines into multiple levels based on data types processed:
  - Level 1: vector-only operations (e.g. vector addition)
  - Level 2: matrix-vector operations (e.g. matrix-vector multiplication)
  - Level 3: matrix-matrix operations (e.g. matrix multiplication)

cuBLAS Idiosyncracies

- For legacy compatibility, cuBLAS operates on column-major matrices



- cuBLAS also has a legacy API which was dropped since CUDA 4.0, this lecture will use the new cuBLAS API
  - If you find cuBLAS code that doesn't quite match up, you may be looking at the old cuBLAS API

cuBLAS Data Management

- Device memory in cuBLAS is allocated as you're used to:
  `cudaMalloc`

- Transferring data to/from the device uses cuBLAS-specific functions:
  - `cublasGetVector/cublasSetVector`
  - `cublasGetMatrix/cublasSetMatrix`

cuBLAS Data Management

- Example:

```
cublasStatus_t cublasSetVector(int n,
int elemSize, const void *x, int incx,
void *y, int incy);
```

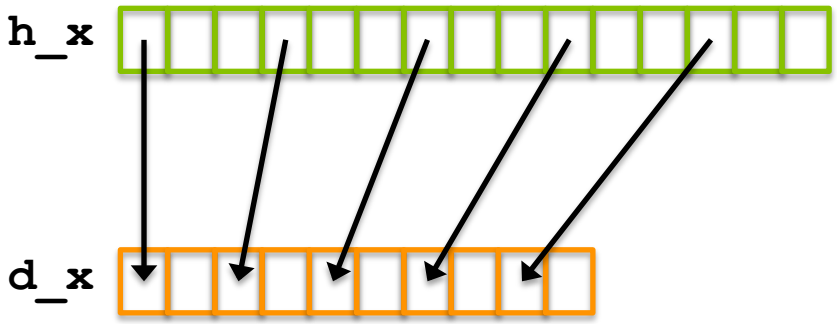
where:

- **n** is the number of elements to transfer to the GPU
- **elemSize** is the size of each element (e.g. sizeof(int))
- **x** is the vector on the host to copy from
- **incx** is a stride in **x** of the array cells to transfer to
- **y** is the vector on the GPU to copy to
- **incy** is a stride in **y** of the array cells to transfer to

cuBLAS Data Management

- Example:

```
cublasSetVector(5, sizeof(int), h_x, 3,
d_x, 2);
```



cuBLAS Data Management

- Similarly:

```
cublasStatus_t cublasSetMatrix(int rows,
int cols, int elemSize,
    const void *A, int lda, void *B, int
ldb);
```

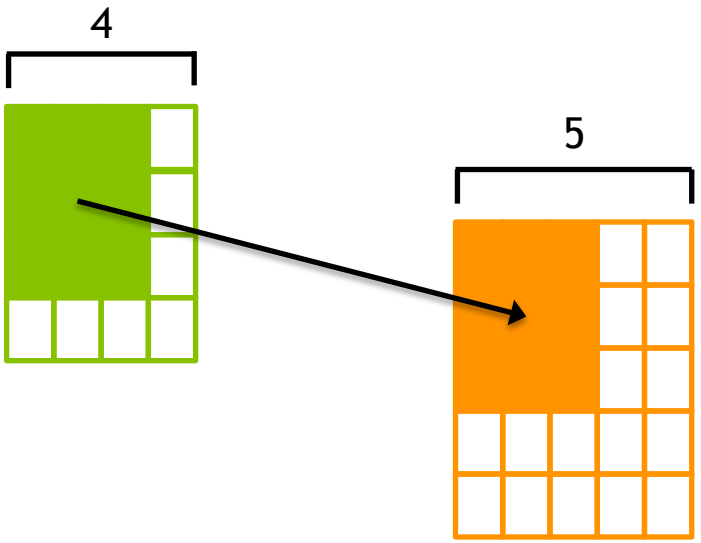
where:

- **rows** is the number of rows in a matrix to copy
- **cols** is the number of cols in a matrix to copy
- **elemSize** is the size of each cell in the matrix (e.g. sizeof(int))
- **A** is the source matrix on the host
- **lda** is the number of rows in the underlying array for **A**
- **B** is the destination matrix on the GPU
- **ldb** is the number of rows in the underlying array for **B**

cuBLAS Data Management

- Similarly:

```
cublasSetMatrix(3, 3, sizeof(int), h_A,
4, d_A, 5);
```



cuBLAS Example

- Matrix-vector multiplication
  - Uses 6 of the 10 steps in the common library workflow:

1. Create a cuBLAS handle using `cublasCreateHandle`
2. Allocate device memory for inputs and outputs using `cudaMalloc`
3. Populate device memory using `cublasSetVector`, `cublasSetMatrix`
4. Call `cublasSgemv` to run matrix-vector multiplication on the GPU
5. Retrieve results from the GPU using `cublasGetVector`
6. Release CUDA and cuBLAS resources using `cudaFree`, `cublasDestroy`

cuBLAS Example

- You can build and run the example `cublas.cu`:

```
cublasCreate(&handle);
cudaMalloc((void **)&dA, sizeof(float) * M * N);
cudaMalloc((void **)&dX, sizeof(float) * N);
cudaMalloc((void **)&dY, sizeof(float) * M);

cublasSetVector(N, sizeof(float), X, 1, dX, 1);
cublasSetVector(M, sizeof(float), Y, 1, dY, 1);
cublasSetMatrix(M, N, sizeof(float), A, M, dA, M);

cublasSgemv(handle, CUBLAS_OP_N, M, N, &alpha, dA, M, dX, 1,
&beta, dY, 1);

cublasGetVector(M, sizeof(float), dY, 1, Y, 1);

/* for sgemm */
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, matrix_size.uiWB, matrix_size.uiHA,
matrix_size.uiWA, &alpha, d_B, matrix_size.uiWB, d_A, matrix_size.uiWA, &beta, d_C,
matrix_size.uiWA)
```

cuBLAS Portability

- Porting to cuBLAS from BLAS is a straightforward process. In general, it requires:
  - Adding device memory allocation/freeing (`cudaMalloc`, `cudaFree`)
  - Adding device transfer functions (`cublasSetVector`, `cublasSetMatrix`, etc)
  - Transform library routine calls from BLAS to cuBLAS (e.g. `cblas_sgemv` ➔ `cublasSgemv`)

cuBLAS Portability

- Some common optimizations following a naive BLAS ➔ cuBLAS port are:
  - Reusing device memory allocations
  - Removing redundant data transfers from and to the device
  - Adding streamed execution using `cublasSetStream`

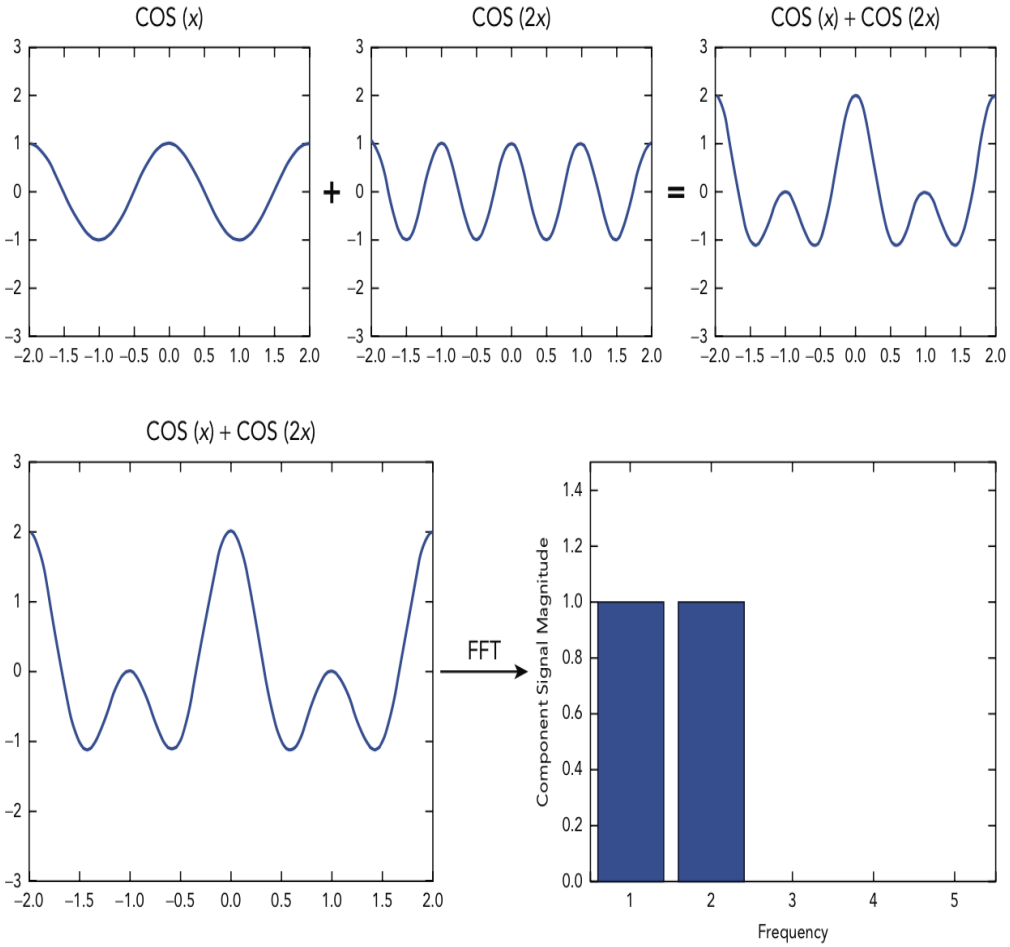
cuBLAS Summary

- cuBLAS makes accelerating legacy BLAS applications simple and easy
  - Very little added code
  - Straightforward mapping from BLAS routines to cuBLAS routines
  - Flexible API improves portability

- For new linear algebra applications, cuBLAS offers a high-performance alternative to BLAS
  - High-performance kernels with very little programmer time

cuFFT

- cuFFT offers an optimized implementation of the fast Fourier transform



cuFFT Configuration

- In cuFFT terminology, plans == handles
  - cuFFT plans define a single FFT transformation to be performed

- cuFFT uses plans to derive the internal memory allocations, transfers, kernels required to implement the desired transform

- Plans are created with:

```
cufftResult cufftPlan1d(cufftHandle *plan, int nx, cufftType
type, int batch);
cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny,
cufftType type);
cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int
nz, cufftType type);
```

cuFFT Configuration

- `cufftType` refers to the data types of a transformation, for example:
  - Complex-to-complex: `CUFFT_C2C`
  - Real-to-complex: `CUFFT_R2C`
  - Complex-to-real: `CUFFT_C2R`

cuFFT Example

- A complex-to-complex 1D cuFFT plan and executing it, using 6 of the 10 steps in the common library workflow:

1. Create and configure a cuFFT plan
2. Allocate GPU memory for the input samples and output frequencies using `cudaMalloc`
3. Populate GPU memory with input samples using `cudaMemcpy`
4. Execute the plan using a `cufftExec*` function
5. Retrieve the calculated frequencies from GPU memory using `cudaMemcpy`
6. Release CUDA and cuFFT resources using `cudaFree`, `cufftDestroy`

cuFFT Example

- You can build and run an example `cufft.cu`:

```
cufftPlan1d(&plan, N, CUFFT_C2C, 1);

cudaMalloc((void **)&dComplexSamples, sizeof(cufftComplex) *
N);

cudaMemcpy(dComplexSamples, complexSamples,
sizeof(cufftComplex) * N, cudaMemcpyHostToDevice);

cufftExecC2C(plan, dComplexSamples, dComplexSamples,
CUFFT_FORWARD);

cudaMemcpy(complexFreq, dComplexSamples, sizeof(cufftComplex) *
N, cudaMemcpyDeviceToHost);
```

cuFFT Summary

- Like cuBLAS, cuFFT offers a high-level and usable API for porting legacy FFT applications or writing new ones
  - cuFFT's API is deliberately similar to industry-standard library FFTW to improve programmability
  - Offers higher performance for little developer effort

Drop-In CUDA Libraries

- Drop-In CUDA Libraries allow seamless integration of CUDA performance with existing code bases
  - Full compatibility with industry-standard libraries, expose the same external APIs
  - BLAS → NVBLAS
  - FFTW → cuFFTW

- Two ways to use Drop-In Libraries:
  - Re-link to CUDA Libraries
  - LD_PRELOAD CUDA Libraries before their host equivalents

Drop-In CUDA Libraries

- Re-linking legacy applications to CUDA Libraries:
  - Suppose you have a legacy application that relies on BLAS:

```
$ gcc app.c -lblas -o app
```

  - Recompiling with NVBLAS linked will automatically accelerate all BLAS calls

```
$ gcc app.c -lnvblas -o app
```

- Alternatively, simply set `LD_PRELOAD` when executing the application:

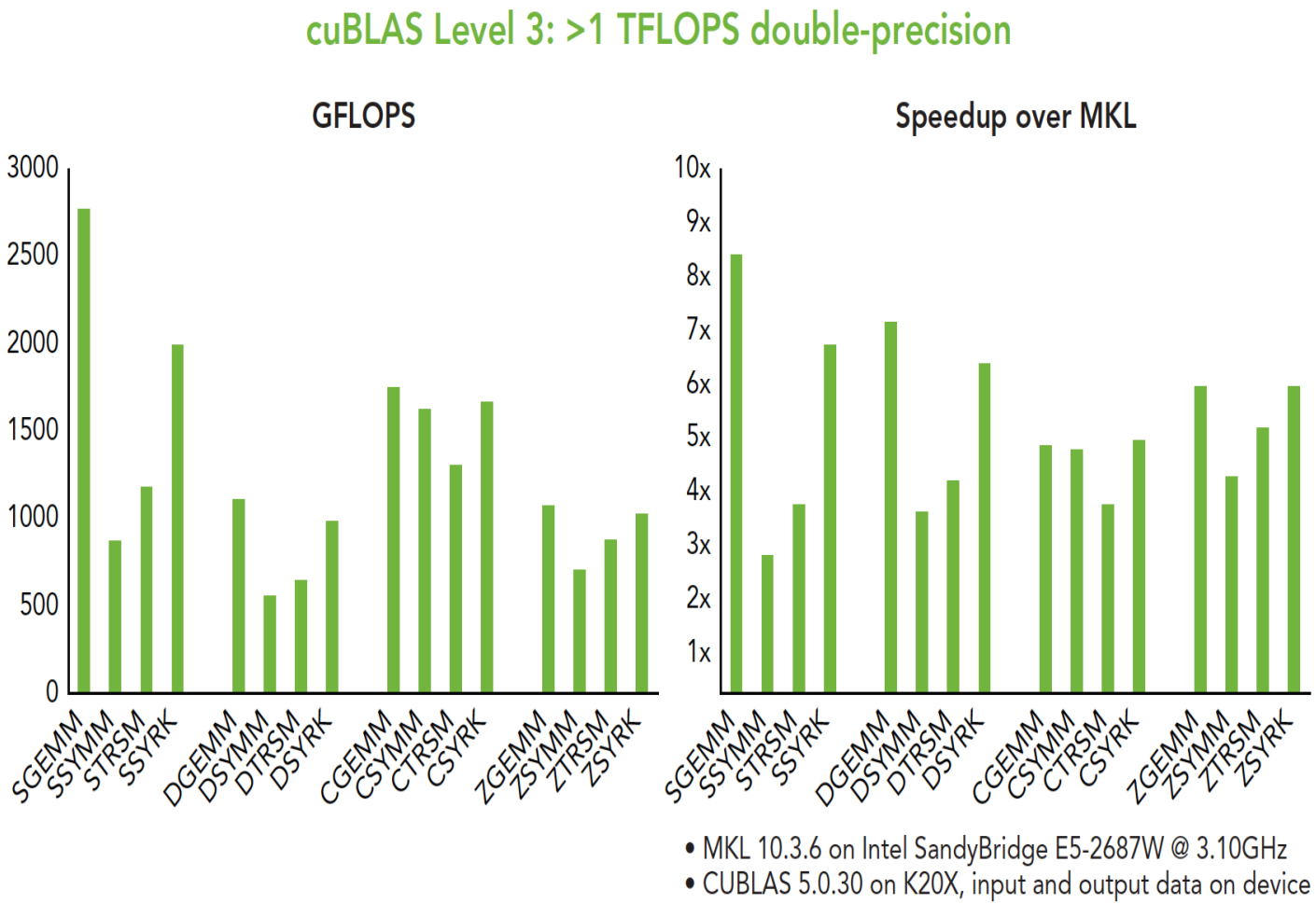
```
$ env LD_PRELOAD=libnvblas.so ./app
```

Survey of CUDA Library Performance

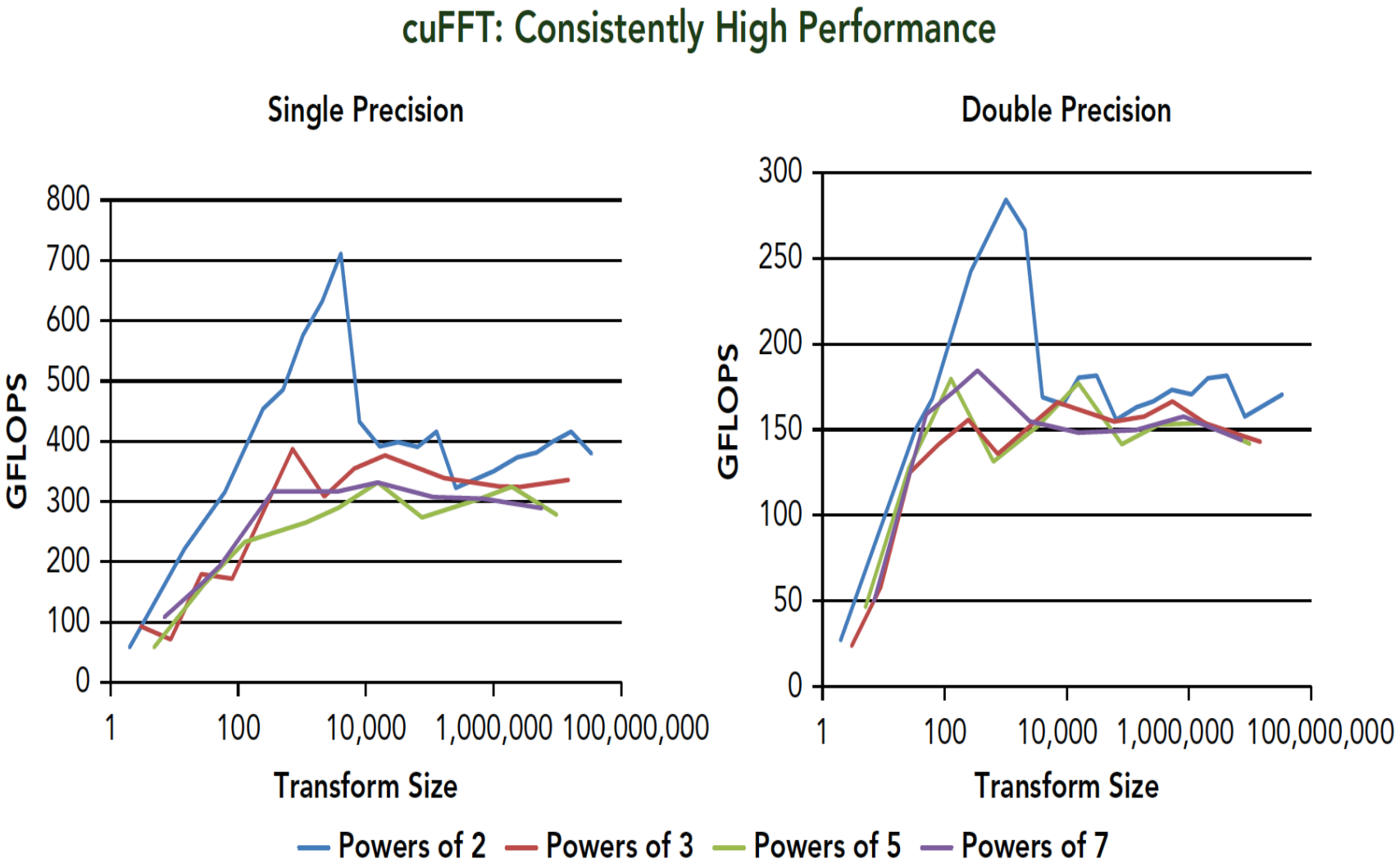
- We've seen that cuBLAS and cuFFT are high-level, programmable libraries (like their host counterparts)
  - No CUDA-specific concepts (e.g. thread blocks, pinned memory, etc)

- Let's do a brief survey of CUDA Library performance to see the performance improvements possible
  - Focus on the same libraries (cuBLAS and cuFFT) but similar data on other libraries is available in the book and online

Survey of CUDA Library Performance



Survey of CUDA Library Performance



Suggested Readings

1. All sections in Chapter 8 of *Professional CUDA C Programming* except *Using OpenACC*
2. *cuSPARSE User Guide*. 2014. http://docs.nvidia.com/cuda/cusparse/
3. *cuBLAS User Guide*. 2014. http://docs.nvidia.com/cuda/cublas/
4. *cuRAND User Guide*. 2014. http://docs.nvidia.com/cuda/curand/
5. *cuFFT User Guide*. 2014. http://docs.nvidia.com/cuda/cufft/
6. *CUDA Toolkit 5.0 Performance Report*. 2013. http://on-demand.gputechconf.com/ gtc-express/2013/presentations/ cuda--5.0-math-libraries-performance.pdf

Manycore GPU Architectures and Programming: Outline

- Introduction
  - GPU architectures, GPGPUs, and CUDA
- GPU Execution model
- CUDA Programming model
- Working with Memory in CUDA
  - Global memory, shared and constant memory
- Streams and concurrency
- CUDA ~~instruction intrinsic~~ and library

☛ **Performance, profiling, debugging, and error handling**

- Directive-based high-level programming model
  - OpenACC and OpenMP

GPU Parallelization

- A many-faceted process
  - Performance varies dramatically depending on the implementation of the same algorithms
    - Naïve to highly optimized version

- Many types of optimizations for GPUs
  - Shared memory
  - Constant memory
  - Global memory access patterns
  - Warp shuffle instructions
  - Computation-communication overlap
  - CUDA compiler flags, e.g. loop unrolling, etc
  - Increasing parallelism
  - …

Optimization Opportunities

- Kernel-level optimization:
  - Exposing Sufficient Parallelism
  - Optimizing Memory Access
  - Optimizing Instruction Execution

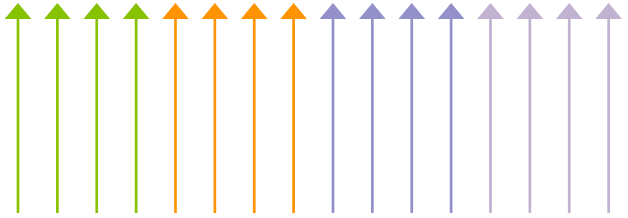
- Host-GPU optimization
  - E.g. kernel and data transfer overlap using CUDA streams

- Profile-driven optimization improves optimizations selection

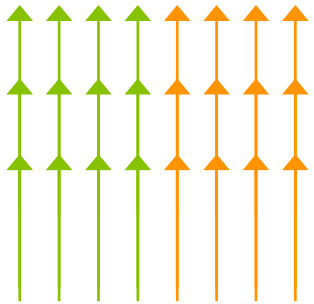
Kernel-Level Optimization

- Exposing Sufficient Parallelism
  - Increase the amount of concurrent work on the GPU so as to saturate instruction and memory bandwidth

- Can be accomplished by:
  1. More concurrently active warps per SM
  2. More independent work assigned to each thread



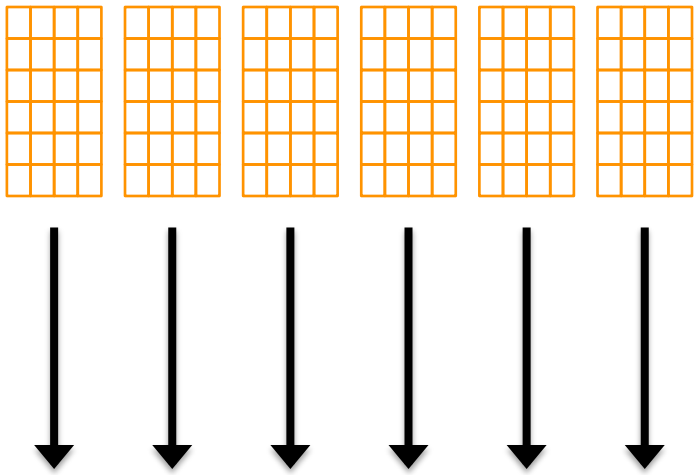
Warp-Level Parallelism



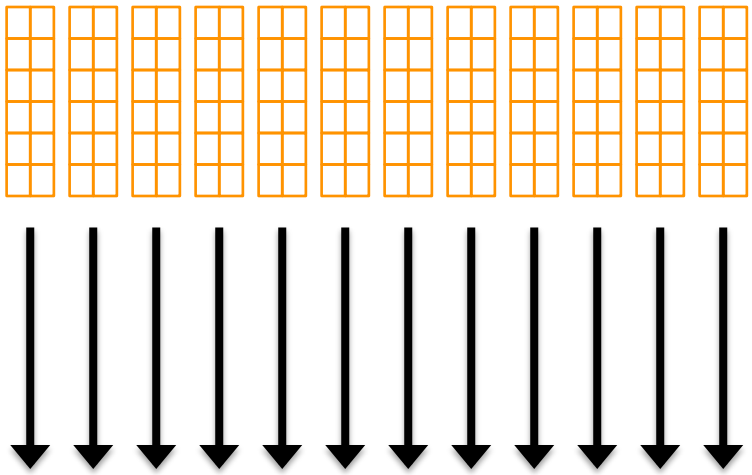
Instruction-Level Parallelism

Kernel-Level Optimization

- Increasing the number of warps per SM/thread block does not guarantee performance improvement
  - Result in fewer per-SM resources assigned to each thread (e.g. registers, shared memory)



Less parallelism, more per-thread resources



More parallelism, smaller per-thread resources

Kernel-Level Optimization

- Creating more independent work per thread
  - loop unrolling or other code transformations that expose instruction-level parallelism,
  - But may also increase per-thread resource requirements

```
int sum = 0;
for (int i = 0; i < 4; i++)
{
    sum += a[i];
}
```

Requires 2 registers (`sum`, `i`), no instruction-level parallelism

```
int i1 = a[0];
int i2 = a[1];
int i3 = a[2];
int i4 = a[4];
int sum = i1 + i2 + i3 + i4;
```

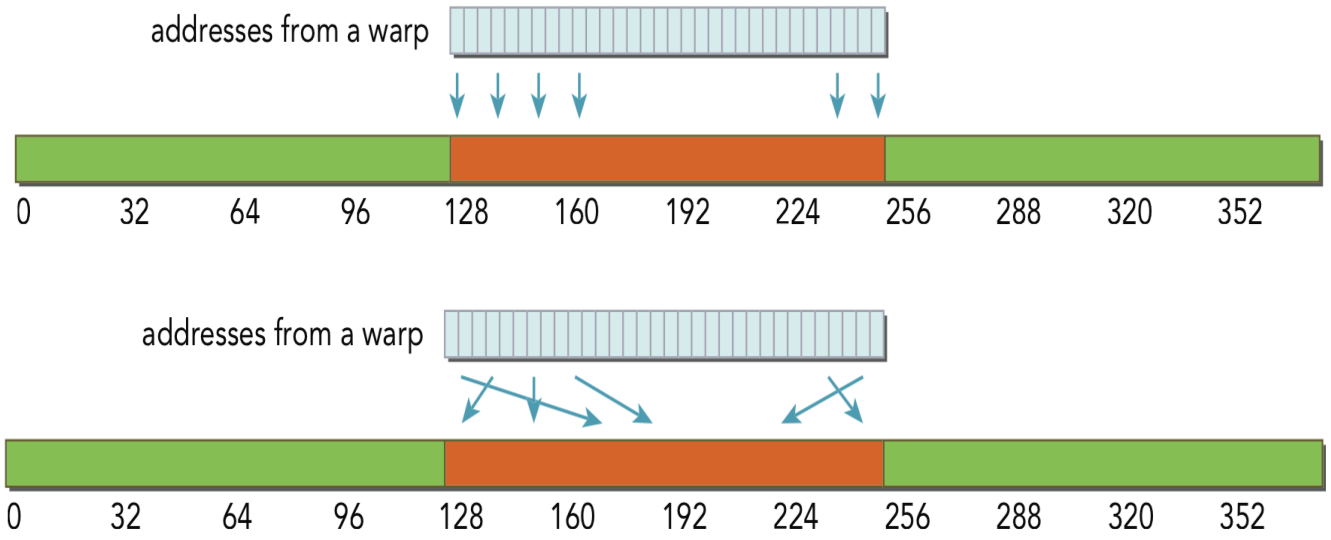
Requires 5 registers (`i1`, `i2`, `i3`, `i4`, `sum`), four-way instruction-level parallelism

Kernel-Level Optimization

- Optimizing memory access to maximize:
  - Memory bandwidth utilization (efficiency of memory access patterns)
  - Memory access concurrency (sufficient memory requests to hide memory latency)

Kernel-Level Optimization

- Aligned, coalesced global and shared memory accesses optimize memory bandwidth utilization
  - Constant memory prefers a broadcast access pattern



Kernel-Level Optimization

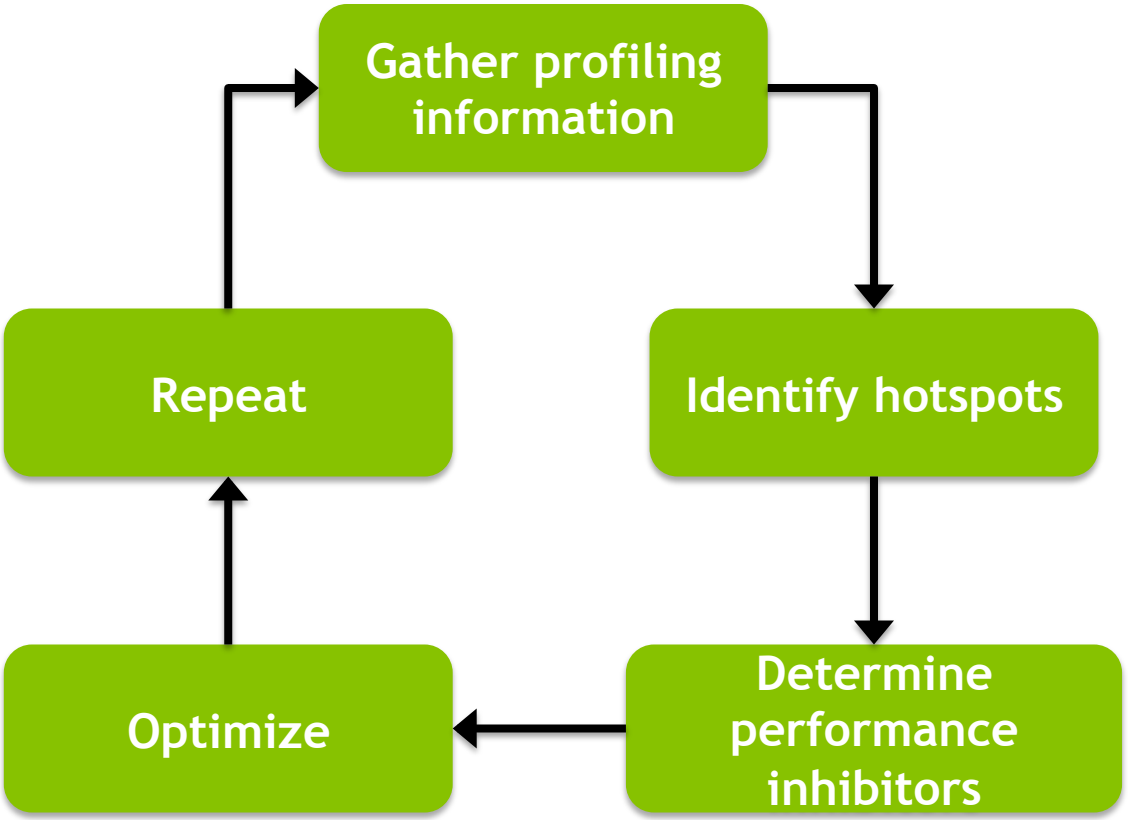
- Optimizing Instruction Execution focuses on:
  - Hiding instruction latency by keeping a sufficient number of warps active
  - Avoiding divergent execution paths within warps
    - If inside a kernel

- Experimenting with thread execution configuration can produce unexpected performance gains from more or less active warps

- Divergent execution within a warp produces reduced parallelism as warp execution of multiple code paths is serialized

Profile-Driven Optimization

- Profile-driven optimization is an iterative process to optimize program based on **quantitative** profiling info
  - As we apply optimization techniques, we analyze the results using `nvprof` and decide if they are beneficial



Profile-Driven Optimization

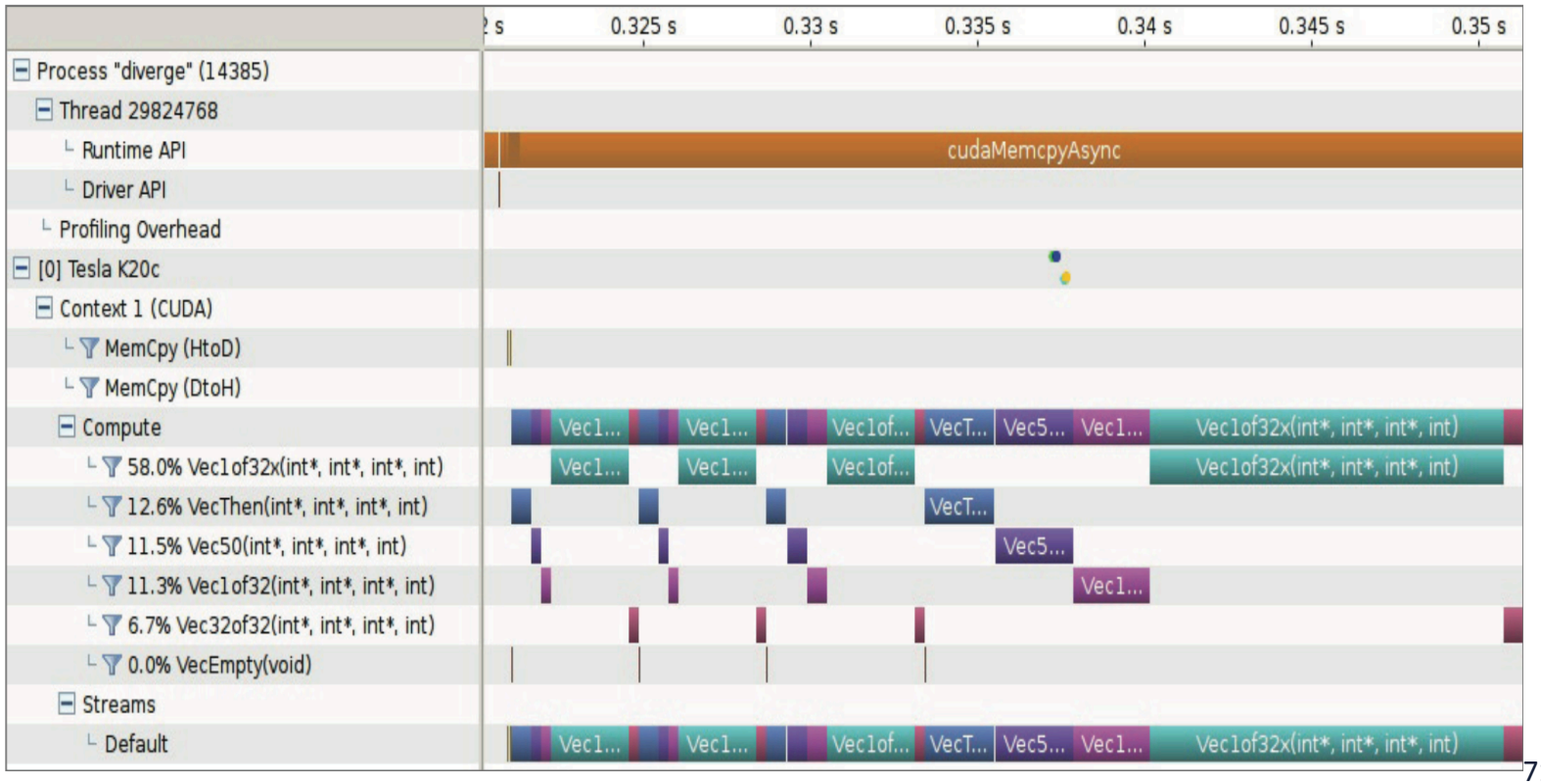
- The key challenge in profile-driven optimization is to determine performance inhibitors in hotspots
  - `nvvp` and `nvprof` are invaluable tools for this

Profile-Driven Optimization

- `nvprof` profiling modes:
  - *Summary Mode*: default mode, displays execution time information on high-level actions such as kernels or data transfers
  - *Trace Mode*: Provides a timeline of CUDA events or actions in chronological order
  - *Event/Metric Summary Mode*: Aggregates event/metric counts across all kernel invocations
  - *Event/Metric Trace Mode*: Displays event/metric counts for each kernel invocation

Profile-Driven Optimization

- The NVIDIA Visual Profiler (nvvp) is also a powerful tool for guiding profile-driven optimization
  - Offers a number of views to inspect different parts of a CUDA application



CUDA Debugging

- An important part of CUDA software development is the ability to debug CUDA applications

- CUDA offers a number of debugging tools, split into two categories:
  - Kernel Debugging
  - Memory Debugging

CUDA Debugging

- Kernel Debugging tools help us to analyze the correctness of running CUDA kernels by inspecting running application state

- Memory Debugging Tools help us detect application bugs by observing irregular or out-of-bound memory accesses performed by CUDA kernels

CUDA Kernel Debugging

- Primary tool for the job: **cuda-gdb**
  - Intentionally built to be similar to the host debugging tool gdb
  - Requires compilation with special flags to be useful:

```
$ nvcc –g –G foo.cu -o foo
```

- Once an application is compiled in debug mode, running it under **cuda-gdb** is possible using:

```
$ cuda-gdb foo
...
(cuda-gdb)
```

CUDA Kernel Debugging

- `cuda-gdb` uses most of the same commands as `gdb`

- One main difference is the idea of CUDA Focus, or the current thread that `cuda-gdb` is focused on and against which all commands run
  - Query the current focus using:

```
(cuda-gdb) cuda thread lane warp block
sm grid device kernel
```

  - Example of setting focus to the 128th thread in the current block:

```
(cuda-gdb) cuda thread (128)
```

CUDA Kernel Debugging

- `printf` is another form of CUDA Kernel Debugging
  - Only available on devices of compute capability 2.0 or higher

- Prints are buffered on the device and periodically transferred back to the host for display
  - Size of this buffer configurable with `cudaSetDeviceLimit`

- Buffer contents are transferred to the host after any CUDA kernel launch, any host-side explicit synchronization, any synchronous memory copies

CUDA Memory Debugging

- Memory Debugging detects memory errors in CUDA kernels that are likely indicative of bugs in the code
  - For example: out-of-bounds memory accesses

- There is a single tool for Memory Debugging, `cuda-memcheck`, which contains two utilities:
  - The `memcheck` tool
  - The `racecheck` tool

CUDA Memory Debugging

- The compilation process for `cuda-memcheck` is more involved than for `cuda-gdb`
  - Building with full debug options affects performance, which may make memory errors harder to hit
  - Applications should always be compiled with `-lineinfo`
  - Applications should also be compiled to include symbol information, but doing this varies by platform
    - Linux: -Xcompiler –rdynamic
    - Windows: -Xcompiler /Zi
    - …

CUDA Memory Debugging

- Once the application is compiled, `memcheck` can be used to check for 6 different types of memory errors:
  1. Memory Access Error: Out-of-bounds or misaligned memory access
  2. Hardware Exception: Error reported by hardware
  3. `malloc`/`free` Errors: Improper use of CUDA dynamic memory allocation
  4. CUDA API Errors: Any error return code from a CUDA API call
  5. cudaMalloc Memory Leaks: `cudaMalloc` allocations that are not `cudaFree`'d
  6. Device Heap Memory Leaks: Dynamic memory allocations that are never freed

CUDA Memory Debugging

- The two `cuda-memcheck` utilities offer very different capabilities:
  - **`memcheck`** performs a wide range of memory correctness checks
  - **`racecheck`** verifies that `__shared__` memory usage is correct in an application, a particularly difficult task to perform manually

- `cuda-memcheck` offers a more automated approach to debugging than `cuda-gdb`

CUDA Error Handling

- Proper error handling is an important part of robust CUDA deployment
  - Every CUDA function returns an error code that must be checked
  - If asynchronous operations are used, this error may be a result of a different asynchronous operation failing
  - Return code of `cudaSuccess` indicates success

- CUDA also offers a number of error-handling functions

CUDA Error Handling

```
cudaError_t cudaGetLastError();
```

- Retrieve the latest CUDA error, clearing the CUDA runtime's internal error state to be `cudaSuccess`

```
cudaError_t cudaPeekLastError();
```

- Retrieve the latest CUDA error, but do not clear the CUDA runtime's internal error state

```
const char *cudaGetErrorString(cudaError_t error);
```

- Fetch a human-readable string for the provided error

Suggested Readings

1. Chapter 10 in *Professional CUDA C Programming*
2. Adam DeConinck. *Introduction to the CUDA Toolkit as an Application Build Tool*. GTC 2013. http://on-demand.gputechconf.com/gtc/2013/webinar/cuda-toolkit-as-build- tool.pdf
3. Sandarbh Jain. *CUDA Profiling Tools*. GTC 2014. http://on-demand.gputechconf.com/ gtc/2014/presentations/S4587-cuda-profiling-tools.pdf
4. Thomas Bradley. *GPU Performance Analysis and Optimization*. 2012. http://people.maths .ox.ac.uk/gilesm/cuda/lecs/NV_Profiling_lowres.pdf
5. Julien Demouth. *CUDA Optimization with NVIDIA Nsight(TM) Visual Studio Edition: A Case Study*. GTC 2014. http://on-demand.gputechconf.com/gtc/2014/presentations/S4160- cuda-optimization-nvidia-nsight-vse-case-study.pdf