Introduction to Numerical General Purpose GPU Computing with NVIDIA CUDA

Part II

CUDA C/C++

Language Overview and Programming Techniques

Outline

technische universität
dortmund

- GPU-Helloworld
- CUDA C/C++ Language Overview (with simple examples)
- The nvcc compiler
    - ► Integration of CUDA code into existing projects
- Debugging (return codes, printf, cuda-memcheck, cuda-gdb)
- Intermediate Example: Heat Transfer
- Atomic Operations
- Memory Transfer (Pinned memory, Zero-Copy host memory)
- CUDA accelerated libraries:

CUDA Resources

- Documentation
  - Local: in /sfw/cuda/7.5/doc/pdf
    - CUDA_C_Programming_Guide.pdf
    - CUDA_C_Getting_Started.pdf
    - CUDA_C_Toolkit_Release.pdf
  - Online CUDA API Reference:
  - http://docs.nvidia.com/cuda/cuda-runtime-api/index.html
  - CUDA Toolkit Download for personal installation:
  - https://developer.nvidia.com/cuda-downloads

CUDA API

- API = application programming interface
  - Set of functions for various GPU tasks:
    - Memory allocation
    - Error checking
    - Host-Device synchronization
    - …
  - Functions included in C-style:
    - #include <cuda_runtime.h>
    - Link with –lcudart (not neccessary if using nvcc)
  - Basic functions covered in this course

CUDA Hello World!(I)

```
#include <stdio.h>
// Cuda supports printf in kernels for // hardware with compute compatibility
>= 2.0

__global__ void helloworld()
{
  // CUDA runtime uses device overloading
  // for printf in kernels
  printf("Hello world!\n");
}

int main(void)
{
  helloworld<<<1,1>>>();
  return 0;
}
```

CUDA Hello World – The kernel

```
__global__ void helloworld()
{
  printf("Hello world!\n");
}
```

- Kernel:
  - A function with the *type qualifier* __global__
  - Executed on the GPU
  - *void* return type is required

CUDA Hello World – The launcher

```
int main(void)
{
  helloworld<<<1,1>>>();
  return 0;
}
```

- Executes on the host
- Kernel call followed by <<< x,y >>> syntax
- User-defined C/C++ function (here: main)
- Passes arguments to kernel (here: void)
- Common design in GPU accelerated codes:
  - ► Launcher/Kernel pairs
- Compilation:

  nvcc helloworld.cu –o helloworld

CUDA Hello World – Problems

- No output written
- Application terminated before GPU code execution
- Asynchronous kernel launch
- Host needs to wait for the device to finish
- Synchronization: *cudaDeviceSynchronize*()
  - ► Blocks until the device has completed preceding tasks

```
int main(void)
{
  helloworld<<<1,1>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Simple Programm Vector Addition

vector_add.cu

```
#include <stdio.h>
#include <cuda_runtime.h>
#define N 10
__global__ void add(int *a, int *b, int *c) {
 int tid = blockIdx.x;
 if (tid < N)
  c[tid] = a[tid] + b[tid];
 }
int main(void) {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;
 cudaMalloc((void**)&dev_a, N * sizeof(int));
 cudaMalloc((void**)&dev_b, N * sizeof(int));
 cudaMalloc((void**)&dev_c, N * sizeof(int));
 for (int i = 0; i < N; ++i) {
  a[i] = -i;
  b[i] = i * i;
 }
 cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice);
 add<<<N,1>>>(dev_a, dev_b, dev_c);
 cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);
 cudaFree(dev_a);
 cudaFree(dev_b);
 cudaFree(dev_c);
 return 0;
}
```

GPU kernel vs CPU kernel

GPU

```
__global__ void add(int *a, int *b, int *c)
{
 int tid = blockIdx.x;
 if (tid < N)
  c[tid] = a[tid] + b[tid];
}
/*
Other code
*/
add<<<N,1>>>(dev_a, dev_b, dev_c);
```

Launches the kernel with N=10 blocks with 1 thread per block. Threads are executed in parallel.

CPU

```
#define N 10

void add(int *a, int *b, int *c)
{
  for (int i=0; i < N; ++i)
   c[i] = a[i] + b[i];
}
```

Loop is executed N=10 times in a serial manner.

Getting Information on the GPU

gpu_properties.cu

```
--- General Information for device 0 ---
Device 0: GeForce GTX 980 Ti
CUDA capability Major.Minor version: 5.2
Total global mem: 6143 MBytes (6441730048 bytes)
GPU Max Clock rate: 1190 MHz (1.19 GHz)
Memory Clock rate: 3505 Mhz
Memory Bus Width: 384-bit
Total constant memory: 65536 bytes
Shared memory per block: 49152 bytes
Registers per block: 65536
Warp size: 32
Max memory pitch: 2147483647 bytes
Texture Alignment: 512 bytes
Multiprocessor count: 22
Max threads per block: 1024
Max thread block dimensions (x,y,z): (1024, 1024, 64)
Max grid dimensions (x,y,z): (2147483647, 65535, 65535)
Concurrent copy and kernel execution: Enabled
Run time limit on kernels : Enabled
```

CUDA C/C++ language components

- Host side language support
  - C/C++ standard which is supported by the host compiler
- Host side language extensions
  - Launcher syntax <<<...,...>>>

```
add<<<N,1>>>(dev_a, dev_b, dev_c);
```

Host side language extensions:

- Predefined short vector types
  - Float4, char4, etc.
  - Constructors: make_int2(int x, int y)
  - Available in host and device code
  - Access components by .x, .y, .xy, .xyz, etc.
- See CUDA Toolkit Documentation:
  - http://docs.nvidia.com/cuda/nvrtc/index.html#predefined-types

Device side language support

- C99:
  - Full support
- C++03
  - C++ features:
  - Classes
  - Derived classes (no virtual member functions)
  - Class and function templates
  - No rtti
  - No exception handling
  - No STL

Device side language support

- Since CUDA 7.0: C++11
  - auto: deduction of a type from initializer
  - initializer lists
  - ranged-based for loops
  - Lambda functions
- Full Details:
  - Appendix E of programming guide

```
__global__ void add(int *a, int *b, int *c)
```

Devide side: function type qualifiers

- __global__
  - Declares a kernel
  - GPU function that can be called from the host
  - For compute capability 3.5:
    Callable from device too
  - Has to be declared void
  - No recursion
  - No variable number of arguments

Devide side: built-in variables

- gridDim: dim3, dimension of current grid
- blockIdx: unit3, block index in current grid
- blockDim: dim3, dimenions of current block
- threadIdx: uint3, thread index in current block
- warpSize: int, size of a warp

```
__global__ void add(int *a, int *b, int *c)
{
 int tid = blockIdx.x;
 if (tid < N)
  c[tid] = a[tid] + b[tid];
}
```

Devide side: function type qualifiers

- __device__
  - Declares a function callable from device only
  - Recursion supported for compute capability >= 2.x
  - For compute capability 3.5:
  - Has to be declared void
  - No recursion
  - No variable number of arguments

```
__device__ int iadd(int a, int b)
{
  return a+b;
}
```

```
__global__ void add(int *a, int *b, int *c)
{
  int tid = blockIdx.x;
  if (tid < N)
   c[tid] = iadd(a[tid], [tid]);
}
```

Devide side: function type qualifiers

- __host__
  - ►Can be called from host only
  - ►Default unqualified behavior
- __device__ __host__
  - ►creates device and host version

```
__host__  __device__  float ceilf ( float x ) __host__  __device__
float cosf ( float x )
```

Devide side: variable type qualifiers

- __device__

```
__device__ float pi = 3.141592;
```

  - Variable in global device memory
  - Lifetime of application
  - Accessible by all threads all the time:
    - Communication
    - Race conditions or deadlocks
    - Needs synchronization or atomic operations (later)
  - Can be accessed from host by API functions:
    - cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()

CUDA Math API

- Always callable from device code (__host__ __device__)
- Similar to <cmath> functions with explicit single/double overloads:
  - sinf(float x) / sin(double x)
  - powf(float x) / pow(double x)
- No includes neccessary when compiling with nvcc

Intrinsics

- (__host __ __device__) functions may need more instructions on the device to meet accuracy requirements
- __device__ qualified functions
- For many standard functions there is an intrinsic function with fewer instructions, but reduced accuracy
- sinf(float x) / __sinf(float x)

Test our knowledge with an example

Addition of long vectors

- vector_add_threads.cu
- vector_add_loop.cu

technische universität dortmund

Addition of long vectors

```
#define N (33 * 1024)

__global__ void add(int *a, int *b, int *c)
{
 int tid = blockIdx.x * blockDim.x + threadIdx.x;
 if (tid < N)
   c[tid] = a[tid] + b[tid];
}
void launcher()
{
  /* other code */
  int threadsPerBlock = 128;
  int blocksPerGrid = (N+threadsPerBlock-1)/threadsPerBlock;
  add<<<blocksPerGrid,threadsPerBlock>>>(dev_a, dev_b, dev_c);
}
```

- Ok, but neither dimension of a grid of blocks may exceed 65535
- With 128 threads we get in trouble for vectors with 65535 * 128 = 8388480 elements

Addition of long vectors: alternative

```
__global__ void add(int *a, int *b, int *c)
{
 int tid = blockDim.x * blockIdx.x + threadIdx.x;
 while (tid < N) {
   c[tid] = a[tid] + b[tid];
   // blockDim.x: number of threads in x-blocks
   // gridDim.x : number of blocks in x-grid
   tid += blockDim.x * gridDim.x;
 }
}
void launcher()
{
  /* other code */
  int threadsPerBlock = 128;
  int blocksPerGrid = 128;
  add<<<blocksPerGrid,threadsPerBlock>>>(dev_a, dev_b, dev_c);
}
```

Works without exceeding grid dimensions

CUDA code can:

- Contain CPU code (host code)
  - Variable declarations, memory allocation, CPU functions
  - Macros, pragmas, defines
- Contain GPU code (device code)
  - __global__ kernel functions
  - __device__ functions
- Contain mixed code
  - Launcher functions with kernel calls
  - CUDA API structures, func<<<x,y>>> syntax

nvcc treats these cases differently:

- Host (CPU) code:
  - Uses a *host compiler* to compile (i.e. gcc)
  - Compiler flags for the host compiler
  - Object files linked by host compiler
- Device (GPU) code:
  - Cannot use host compiler
  - Fails to understand i.e. __global__ syntax
  - Needs nvcc
- Mixed Code:
  - Cannot use host compiler, needs nvcc

nvcc Compiler Design

- Similar usage to standard C/C++ compilers
  - ▶Compiler syntax
  - ▶Flags:
  - ▶Standard flags for generating debug info or optimization: -g, -O3
- Use the host compiler as much as possible
- Compiling CUDA applications is complicated
- Requires more steps to produce the binary

nvcc Compilation Workflow

- nvcc is a compiler wrapper
- nvcc proceeds in separate phases through the compilation process
- Different phases can be executed manually:
  - Compile object files with desired flags
  - Link into an executable
  - Build a library with the libtool of the OS

Phase 1

- Separation into host, device and *mixed* code
- nvcc processes code as C++, not as C

Phase 2: *mixed* code handling

- Launch syntax <<<...,...>>> handling:
  - <<<...,...>>> is a convenience syntax
  - Replace <<<...,...>>> by API calls to set parameters
- Result:
- Intermediate file similar to C++
- Mixed code now passes as host code with API calls and library dependencies

nvcc Compilation Phases

Phase 3: host code

- Takes generated and remaining host code as Input
- nvcc passes code to the host compiler
- Compilation by the host compiler
- Result: regular object files

Phase 4: device code

- Processing of CUDA kernels
- Compile with nvcc into device object files

Phase 5: linking

- Combine host and device object files into an executable
- Uses the linker of the host compiler

Summary

- Simple use of nvcc invokes all five phases
- Split compilation manually by compiler commands
- Check: nvcc –arch=sm_20 helloword.cu –v –dryrun
- http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

nvcc Compiling

- nvcc –cuda vector_add.cu
  - Produces vector_add.cu.cpp.ii
- Resulting file can be compiled by host compiler
- Needs to link the CUDA runtime (cudart)
- Allows use of custom compiler

Adding CUDA code to existing projects and build systems

- Use build system like CMake
- Makefile-based build systems

Integrate CUDA into an existing Makefile project

- Existing source files
- Addition of some CUDA kernels
- No desire to replace all compiler calls by nvcc
- Place launcher declaration in header file(s)

Header file: cuda_extension.h

```
#ifndef __CUDAEXT__
#define __CUDAEXT__
void launcher1(...);
void launcher2(...);
void launcher3(...);
#endif
```

- Write kernels and launchers to a new file
- cuda_extension.cu
- Modify application code slightly:
  - ▶ #include <cuda_runtime.h>
  - ▶ Allows use of CUDA API function calls
  - ▶ #include <cuda_extension.h>

cuda_extension.cu

```
__global__ void kernel1(...)
{
  //kernel code
}
void mylauncher1(...)
{
  // configure kernel launch
  // kernel launch
  // error checking
}
```

  - ▶ Call launcher functions from application code
  - ▶ Link CUDA runtime (cudart)
- Reminder: code is treated as C++, so extern “C“ syntax needs to be used in Fortran or plain C applications

```
nvcc -c cuda_extension.cu -o cuda_extension.o
g++ -c appfile1.cpp -o appfile1.o -I/path/to/cuda/include
g++ -c appfile2.cpp -o appfile2.o -I/path/to/cuda/include
g++ -o app cuda_extension.o appfile1.o appfile2.o -L/path/to/cuda/lib64 -lcudart
```

- Files are compiled separately
- nvcc only for CUDA code
- Relatively easy to integrate into Makefile projects
- Possible problems:
  - Mixes object files from different compilers

Alternative compilation for Makefile Projects

```
nvcc -cuda cuda_extension.cu
g++ -c cuda_extension.cu.cpp.ii -o cuda_extension.o
g++ -c appfile1.cpp -o appfile1.o -I/path/to/cuda/include
g++ -c appfile2.cpp -o appfile2.o -I/path/to/cuda/include
g++ -o app cuda_extension.o appfile1.o appfile2.o -L/path/to/cuda/lib64 -lcudart
```

- Remember: x.cu.cpp.ii files are guaranteed to be compilable by the nvcc host compiler
- -cuda:
  - Replaces launcher syntax <<<...,...>>> by API functions
  - Inlines API headers
  - Generates device binaries
- Result: C++ source that can be compiled by host compiler

__NVCC__ and __CUDACC__

- Test whether file is compiled by nvcc
- Test whether file is regarded as CUDA source
- Use same header for device and host code

__CUDA_ARCH__

- Available only for device code

```
#if __CUDA_ARCH__ >= 130
 // can use double precision
#else
  #error "No double precision available for compute capability < 1.3"
#endif
```

Important compiler flags

-c -o -I -L -l -D -v

- Same as in GCC

-cuda -cubin -ptx -gpu -fatbin -link

- Execute a certain compilation stage

-g -G

- Generate debug info for host + device code

-Xcompiler -Xlinker

- Forward flags to host compiler and linker
  - -Xcompiler=-Wall,-Wno-unused-function

-keep

- Keep intermediate files from various stages
- For debugging purposes

-arch

- Create optimized code for specific compute capability

-arch

- Guaranteed to work on higher compute capabilities
- Usage:
  - -arch=sm_11, -arch=sm_20, -arch=sm_35
- Highly important compiler flag
- Includes future GPUs:
  - If compiled with same major toolkit version
- If not set, the lowest supported instruction set architecture (ISA) is set

Important compiler flags

-arch

- If not set other undesirable effects can happen:
  - No double precision if arch < sm_13
  - No printf in kernels
- Set to something that supports the used features
- rely on PTX compiler in the driver for newer GPUs

GPU Dot Product Example

$$v \cdot w = \sum_{i=1}^{n} v_i \cdot w_i \; for \; v, w \in R^n$$

- Handle pairwise multiplication by threads
- Each thread handles a partial sum
- Join partial sums by a *reduction* operation
  - Needs thread coorperation
- Store intermediate results in *shared memory*
  - On chip low latency memory, shared between threads in a block
  - Used for thread communication
  - Needs synchronization to avoid race conditions

vector_dot.cu

```
const int N = 33 * 1024;
const int threadsPerBlock = 256;
__global__ void dot(float *a, float *b, float *c)
{
  __shared__ float cache[threadsPerBlock];
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int cacheIndex = threadIdx.x;
  float temp = 0;
  while(tid < N) {
   temp += a[tid] * b[tid];
   tid += blockDim.x * gridDim.x;
  }
  cache[cacheIndex] = temp;
  __synchthreads();
  /* remaining code */
}
```

GPU Dot Product Example

technische universität dortmund

```
/* other code */
cache[cacheIndex] = temp;
__syncthreads();
 /* remaining code */
```

- Need to synchronize before joining sums
- __syncthreads:
  - Guarantees that every thread in the block has completed instructions prior to the __syncthreads call
- Can now safely join the partial sums by *reduction*

GPU Dot Product: Reduction

- Reduction:
  - Common operation in parallel computing
- Complexity: proportional to log of the array length
- threadsPerBlock must be a power of 2
- log2(threadsPerBlock) reduction steps

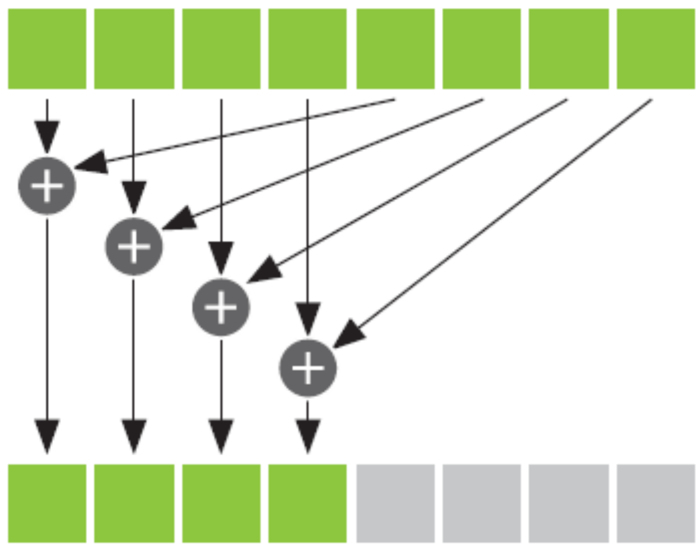


Image: Courtesy of NVIDIA Coorp

GPU Dot Product: Reduction Code

```
/* preceeding code */
cache[cacheIndex] = temp;
__syncthreads();
// Guaranteed: All writes to the shared memory cache finished
// Reduction: threadsPerBlock has to be a power of 2
int i = blockDim.x/2;
while(i != 0) {
 if (cacheIndex < i) {
   cache[cacheIndex] += cache[cacheIndex + i];
 }
 // make sure all writes are finished
 __syncthreads();
 i /= 2;
}
// Store the sum of the blocks in CUDA array accessible from
// host code
if (cacheIndex == 0)
  c[blockIdx.x] = cache[0];
}
```

GPU Dot Product

```
cudaMemcpy( dev_a, a, N*sizeof(float),cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, N*sizeof(float),cudaMemcpyHostToDevice );
dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b, dev_partial_c );
// copy partial sums array from GPU to CPU
cudaMemcpy( partial_c, dev_partial_c, blocksPerGrid*sizeof(float),
cudaMemcpyDeviceToHost );
// add the partial sums on the CPU
float c = 0;
for (int i=0; i<blocksPerGrid; i++)
{
  c += partial_c[i];
}
```

- Compute final result on CPU
- blocksPerGrid=32
- Waste of resources to add 32 numbers on massively parallel hardware

Pitfalls


```
int i = blockDim.x/2;
while(i != 0) {
 if (cacheIndex < i) {
   cache[cacheIndex] += cache[cacheIndex + i];
 }
 __syncthreads();
 i /= 2;
}
```

- **Beware**: Placement of __syncthreads call
- No thread will advance until every thread in the block has executed __syncthreads
- If-clause: *thread divergence*
- Result: *deadlock*

```
int i = blockDim.x/2;
while(i != 0) {
 if (cacheIndex < i) {
   cache[cacheIndex] += cache[cacheIndex + i];
   __syncthreads();
 }
 i /= 2;
}
```

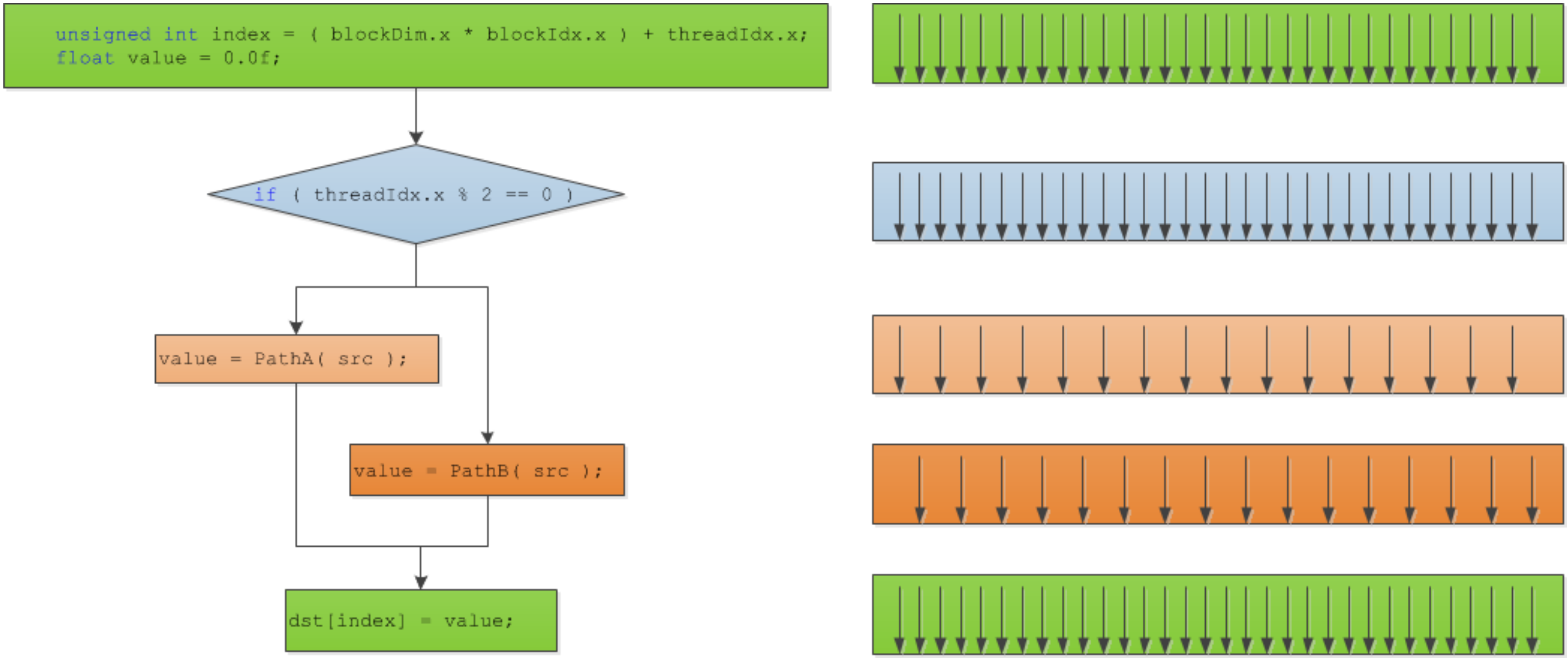


Image: Courtesy Kirk, David B. and Wen-mai, W: Morgan Kaufmann Publishers

- C++-style error checking

```
double *d;
try {
 d = new double[10000000000000];
 } catch (std::bad_alloc &e) {
 std::cerr << e.what() << std::endl;
 }
```

- Error information encoded in exceptions
- Unhandled exection terminate program

- C-style error checking

```
double *d = (double*) malloc(1000000000000*sizeof(double));
if(d == NULL) {
 fprintf(stderror,"memory allocation error");
 exit(1);
}
```

- Functions return error value
- Encode different errors with return values
- Example: malloc
  - ▶Pointer to the allocated memory or NULL

CUDA API error handling

- API calls return a cudaError_t
- Pitfall: kernel launches are an exception
- Pass cudaError_t to an error handling function
- Error handling function identifies exact error

Two options for error checking

- Use a global error checking function
  - ▶ Make function available in a header file
- Use the preprocessor
  - ▶ Can be combined with first option

```
void checkCudaErrors(cudaError_t err, const char *userLabel) {
 if(cudaSuccess != err) {
  fprintf(stderr,
  "checkCudaErrors() Driver API error = %04d \"%s\" at user label \"%s\".\n",
  err, cudaGetErrorString(err), userLabel);
  exit(EXIT_FAILURE);
 }
}
/* other code */
checkCudaErrors(cudaMalloc((void**)&dev_a,1*sizeof(int)),"allocating dev_a");
/* other code */
```

Using the preprocessor

```
#ifndef checkCudaErrors
#define checkCudaErrors(err) __checkCudaErrors(err, __FILE__, __LINE__)
void __checkCudaErrors(cudaError_t err, const char *file, const int line)
{
 if(cudaSuccess != err) {
  fprintf(stderr,
  "checkCudaErrors() Driver API error = %04d \"%s\" from file <%s>, line %i.\n",
  err, cudaGetErrorString(err), file, line);
  exit(EXIT_FAILURE);
 }
}
#endif
/* other code */
int n = 10000000000000;
checkCudaErrors(cudaMalloc((void**)&dev_a, n * sizeof(int)));
 /* other code */
```

Kernel Errors

- CUDA kernel launches are synchronous
  - Recall HelloWorld example
- CPU free to continue while GPU computes
- Kernel launches do not return cudaError_t
- Launch errors, errors inside the kernel are not reported immediately
- A cudaError_t is inserted into the error queue after the kernel finished
  - Kernel launch failures will be reported by a subsequent API call

technische universität dortmund

Hard to determine the faulty kernel

- API calls report an error that does not make sense for the API function
  - ULF = „unspecified launch failure“

Possible approach

- Synchronize after *suspicious* kernel calls
  - cudaDeviceSynchronize()
  - cudaGetLastError()

- Can use traditional printf in kernel
  - Use a minimal example with as little blocks/threads as possible

```
__global__ void test(float *a, float *b, float *c, int n)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid < n) {
   c[tid] = a[tid] + b[tid];
   if(blockIdx.x == 1 && threadIdx.x == 0) {
     printf(" %f + %f = %f \n",a[tid],b[tid],c[tid]);
   }
  }
}
```

- Can use assertion in kernels
  - Needs cc 2.0 or higher
- All following host side API calls return *cudaErrorAssert*

Assertions


```
#include <assert.h>
__global__ void testAssert(void)
{
  int is_one = 1;
  int should_be_one = 0;
  // ok
  assert(is_one);
  // halts kernel execution
  assert(should_be_one);
}
int main(int argc, char* argv[])
{
 testAssert<<<1,1>>>();
 cudaDeviceSynchronize();
 return 0;
}
```

GNU Debugger

- Set breakpoints, step through program
- Inspect and modify variables
- Examine program crash state / segfaults
- Print call stack / backtraces
- Can be attached to running applications

cuda-gdb: GPU variant of gdb

- Included in GPU Toolkit
- Same functionality as CPU version
- The functionality is extended to kernels
- Inspect variable contents by block/thread/etc.
- Breakpoint per thread, warp, block, kernel

Drawbacks

- Breakpoints halt entire GPU
- True for implicit and (segfault) and explicit breakpoint

Consequence

- Halts X-server, machine locked
- Not possible on single-GPU
- CUDA 6, cc 5.0
  - Shutting down X not required

cuda-gdb drawbacks for multi-user systems

- cuda-gdb locks other users X processes

Graphical frontends available

- Alinea DDT
- Eclipse
- Visual Studio

cuda-memcheck

- Equivalent of valgrind for CUDA GPUs
- Included in the toolkit
- Host and device correctness checking
- Synchronizes after every kernel call

Use Cases

- Thousands of threads
- Non-trivial indexing (threads, blocks, grid)
- High probability of memory errors
- Race conditions
- CUDA API (kernel launch errors)
- Hard to detect and debug errors

cuda-memcheck Compiler flags

-G

- Creates full debugging information: line numbers, function symbol name, etc.
- Optimization disabled

-lineinfo

- Only file and line info
- Optimization remains enabled
- Often sufficient

-Xcompiler=-rdynamic

- Insert full symbol names into host backtraces

cuda-memcheck

```
nvcc -G vector_add.cu -o vector_add
cuda-memcheck ./vector_add2
```

```
========= CUDA-MEMCHECK
========= Invalid __global__ read of size 4
========= at 0x000001f8 in /data/warehouse14/rmuenste/code/repo/OpenGL/cuda-intro/debugging/
vector_add.cu:10:add(int*, int*, int*) ========= by thread (0,0,0) in block (10,0,0)
========= Address 0xb06400028 is out of bounds
========= Saved host backtrace up to driver entry point at kernel launch time
========= Host Frame:/usr/lib64/nvidia/libcuda.so.1 (cuLaunchKernel + 0x2cd) [0x15865d]
========= Host Frame:./vector_add [0x1613b]
========= Host Frame:./vector_add [0x30113]
========= Host Frame:./vector_add [0x2ba9]
========= Host Frame:./vector_add [0x2acd]
========= Host Frame:./vector_add [0x2afa]
========= Host Frame:./vector_add [0x29ae]
========= Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1ed5d]
========= Host Frame:./vector_add [0x26f9]
=========
========= Program hit cudaErrorLaunchFailure (error 4) due to "unspecified launch failure" on CUDA API
call to cudaFree.
========= Saved host backtrace up to driver entry point at error
========= Host Frame:/usr/lib64/nvidia/libcuda.so.1 [0x2f31b3]
========= Host Frame:./vector_add [0x3da96]
========= Host Frame:./vector_add [0x29ec]
========= Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1ed5d]
========= Host Frame:./vector_add [0x26f9]
=========
========= ERROR SUMMARY: 5 errors
```

cuda-memcheck leak-check

- Removed cudaFree() in vector_add
- cuda-memcheck –leak-check full
- Detects missing cudaFree() for cudaMalloc()
- Sadly, no line numbers for allocation
- Add *cudaDeviceReset()* at the end of main() to enable leak report

cuda-memcheck leak-check

```
nvcc -G vector_add2.cu -o vector_add2
cuda-memcheck --leak-check full ./vector_add2
```

```
========= CUDA-MEMCHECK =========
Leaked 40 bytes at 0xb06400400
========= Saved host backtrace up to driver entry point at cudaMalloc time
========= Host Frame:/usr/lib64/nvidia/libcuda.so.1 (cuMemAlloc_v2 + 0x17f)
[0x13dc4f]
========= Host Frame:./vector_add2 [0x2dee3]
========= Host Frame:./vector_add2 [0x643b]
========= Host Frame:./vector_add2 [0x3e1df]
========= Host Frame:./vector_add2 [0x28e4]
========= Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1ed5d]
========= Host Frame:./vector_add2 [0x2719]
=========
=========
========= LEAK SUMMARY: 120 bytes leaked in 3 allocations
========= ERROR SUMMARY: 0 errors
```

Texture Memory

- Designed for graphics originally
- Texture memory cached on-chip
- Access is in a specific pattern
  - Low latency
  - No global memory read neccessary
- Many numerical applications have access patterns with spatial locality
  - Finite Difference, Finite Volume, Finite Element, Matrix operations

Texture Memory

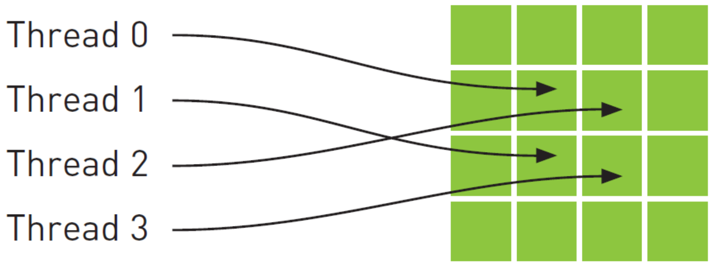
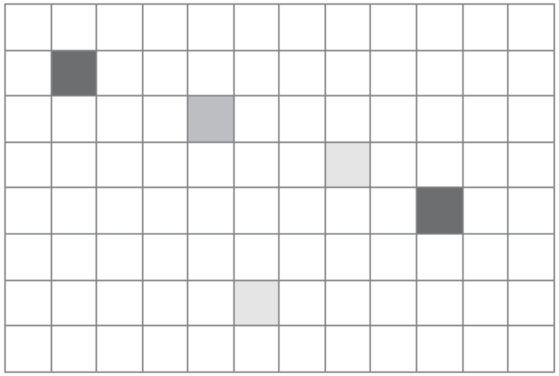


Image: Courtesy NVIDIA Coorp.

- Arithmetically addresses not consecutive
- Would not be cached in typical cashing schemes
- Cashing strategy of CUDA arrays can be modified
- Might achieve same performance as texture memory

Heat Transfer Example



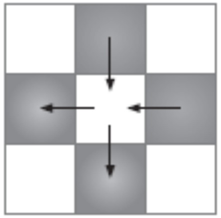


Image: Courtesy NVIDIA Coorp.

- Simplified model
- Basic operations typical for numerical simulations
- Assumptions
  - Rectangular grid of cells
  - Heater cells with constant temperatures
  - Heat flows between cells in every simulation time step

Heat Transfer Example

$$T_{new} = T_{old} + \sum_{n \in Neighbors} k \cdot \left(T_n - T_{old}\right)$$

- New temperature: sum of differences between cell temperature and its neighbors
- k as the ‚flow rate‘
- Only consider the top,left,right,bot neighbors

$$T_{new} = T_{old} + k \cdot \left(T_{top} + T_{bot} + T_{left} + T_{right} - 4 \cdot T_{old}\right)$$

Use a 2D grid of blocks and threads

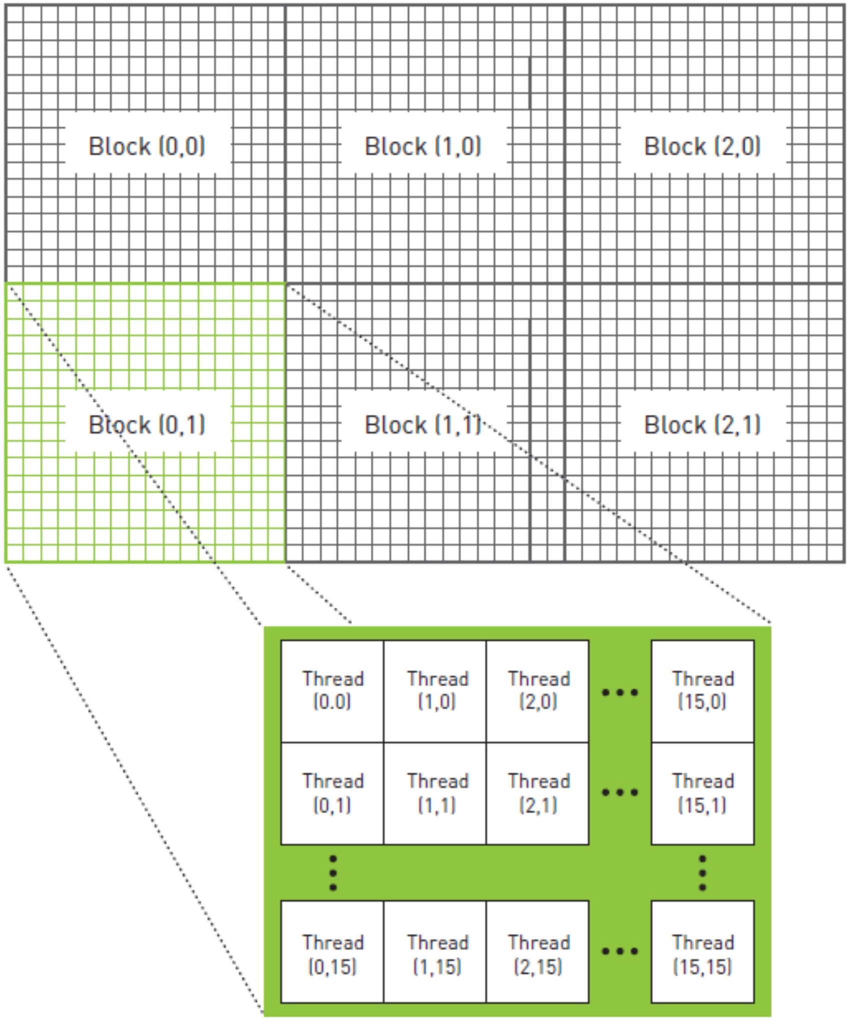


Image: Courtesy NVIDIA Coorp.

Heat transfer algorithm

- Allocate textures for input, output and constant heater values

1. Copy the constant values to input
   - ► copy_const_kernel()
2. Compute output values from input
   - ► blend_kernel()
3. Swap input and output buffers for the next time step

Texture memory setup

- Declare texture reference at global scope
- Allocate a texture buffer
- cudaBindTexture:

```
// Global texture references
// these exist on the GPU
side
texture<float> texConstSrc;
texture<float> texIn;
texture<float> texOut;
```

  - Bind the buffer to a certain texture reference
- Textures reside in texture memory:
  - Need special access function
  - *tex1Dfetch*(textureReference,index)
  - Compiler intrinsic
  - Needs to know arguments at compile time

Texture memory setup

technische universität dortmund

```
// Global texture references
// these exist on the GPU side
texture<float> texConstSrc;
texture<float> texIn;
texture<float> texOut;
/* other code */
struct DataBlock {
 float *dev_inSrc;
 float *dev_outSrc;
 float *dev_constSrc;
 /* other code */
};
 /* other code */
/* allocate memory for texture buffers */
cudaMalloc( (void**)&data.dev_inSrc, imageSize );
cudaMalloc( (void**)&data.dev_outSrc, imageSize );
cudaMalloc( (void**)&data.dev_constSrc, imageSize );
/* bind the buffer to the texture references */
cudaBindTexture( NULL, texConstSrc, data.dev_constSrc, imageSize );
cudaBindTexture( NULL, texIn, data.dev_inSrc, imageSize );
cudaBindTexture( NULL, texOut, data.dev_outSrc, imageSize );
 /* other code */
cudaMemcpy( data.dev_constSrc, temp, imageSize, cudaMemcpyHostToDevice );
cudaMemcpy( data.dev_inSrc, temp, imageSize, cudaMemcpyHostToDevice );
 /* other code */
```

Copy Heater Values

```
__global__ void copy_const_kernel( float *iptr ) {
 // map from threadIdx/BlockIdx to pixel position
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;
 int offset = x + y * blockDim.x * gridDim.x;
 float c = tex1Dfetch(texConstSrc,offset);
 if (c != 0)
  iptr[offset] = c;
 }
```

Simulation loop

technische universität
dortmund

```
bool dstOut = true;
for (int i=0; i<90; i++) {
 float *in, *out;
 if (dstOut) {
  in = d->dev_inSrc;
  out = d->dev_outSrc;
 } else {
  out = d->dev_inSrc;
  in = d->dev_outSrc;
 }
 copy_const_kernel<<<blocks,threads>>>( in );
 blend_kernel<<<blocks,threads>>>( out,
dstOut );
 dstOut = !dstOut;
}
```

Heat Transfer kernel

```
__global__ void blend_kernel( float *dst, bool dstOut ) {
 // map from threadIdx/BlockIdx to grid position
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;
 int offset = x + y * blockDim.x * gridDim.x;
 int left = offset - 1;
 int right = offset + 1;
 if (x == 0) left++;
 if (x == DIM-1) right--;
 int top = offset - DIM;
 int bottom = offset + DIM;
 if (y == 0) top += DIM;
 if (y == DIM-1) bottom -= DIM;
 float t, l, c, r, b;
 if (dstOut) {
  t = tex1Dfetch(texIn,top);
  l = tex1Dfetch(texIn,left);
  c = tex1Dfetch(texIn,offset);
  r = tex1Dfetch(texIn,right);
  b = tex1Dfetch(texIn,bottom);
 } else {
  t = tex1Dfetch(texOut,top);
  l = tex1Dfetch(texOut,left);
  c = tex1Dfetch(texOut,offset);
  r = tex1Dfetch(texOut,right);
  b = tex1Dfetch(texOut,bottom);
 }
 dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}
```

2D Texture Memory


```
texture<float,2> texConstSrc;
texture<float,2> texIn;
texture<float,2> texOut;
__global__ void blend_kernel( float *dst, bool dstOut ) {
 // map from threadIdx/BlockIdx to pixel position
 int x = threadIdx.x + blockIdx.x * blockDim.x;
 int y = threadIdx.y + blockIdx.y * blockDim.y;
 int offset = x + y * blockDim.x * gridDim.x;
 float t, l, c, r, b;
 if (dstOut) {
  t = tex2D(texIn,x,y-1);
  l = tex2D(texIn,x-1,y);
  c = tex2D(texIn,x,y);
  r = tex2D(texIn,x+1,y);
  b = tex2D(texIn,x,y+1);
 } else {
  t = tex2D(texOut,x,y-1);
  l = tex2D(texOut,x-1,y);
  c = tex2D(texOut,x,y);
  r = tex2D(texOut,x+1,y);
  b = tex2D(texOut,x,y+1);
 }
  dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}
```

Example: consider an increment operation

- x++
  - Read value in variable x
  - Add 1 to the value
  - Write the new value back to x
- *read-modify-write* operation
- Can be *tricky* in parallel programming

Atomic Operations

Case A

STEP	EXAMPLE
1. Thread A reads the value in x.	A reads 7 from x.
2. Thread A adds 1 to the value it read.	A computes 8.
3. Thread A writes the result back to x.	x <- 8.
4. Thread B reads the value in x.	B reads 8 from x.
5. Thread B adds 1 to the value it read.	B computes 9.
6. Thread B writes the result back to x.	x <- 9.

Image: Courtesy NVIDIA Coorp.

Case B

STEP	EXAMPLE
Thread A reads the value in x.	A reads 7 from x.
Thread B reads the value in x.	B reads 7 from x.
Thread A adds 1 to the value it read.	A computes 8.
Thread B adds 1 to the value it read.	B computes 8.
Thread A writes the result back to x.	x <- 8.
Thread B writes the result back to x.	x <- 8.

Problem:

- Case A yielded correct result
- Case B incorrect due to scheduling
- Neither A nor B programmed correctly
- We need an uninterrupted *read-modify-write* operation
- Atomic operation

Atomic Operations

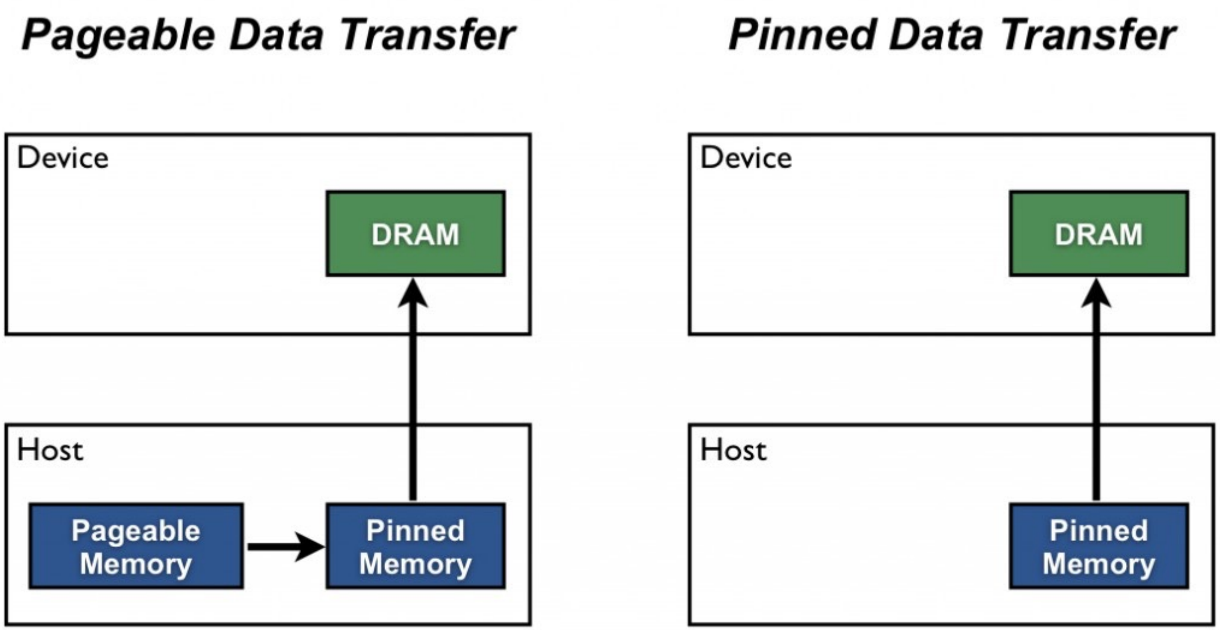
```
__global__ void kernel() {
 __shared__ unsigned int temp[256];
 temp[threadIdx.x] = 0;
 __syncthreads();
 /* other code */
 // int atomicAdd(int* address, int val)
 atomicAdd( &temp[i], 1 );
 /* other code */
 }
```

- Need atomics for various data types
- Available atomics: See Programming Guide
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

Pinned Memory

- malloc: allocates pageable host memory
  - Pageable memory can be swapped to the hard disk by the OS
- cudaHostAlloc: allocates pinned memory
- Alternative name: page-locked memory
  - The OS guarantees that the memory will not be swapped to the disk
- Usage:
  - GPU could use direct memory access (DMA) for copies to and from the host

Pinned Memory



- No staging buffer needed
- About a twofold performance increase can be expected
- Beware:
  - System can run out of memory more quickly without swapping
  - Use it, but use it with care

Bandwidth Benchmark

- Measure copy time and bandwidth
- Use cudaEvent_t to measure time

```
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop );
cudaEventRecord( start, 0 );

kernel<<<blocks,threads>>>( );

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop );
printf( "Elapsed time : %3.1f [ms]\n", elapsedTime );
```

Bandwidth Benchmark

- Measure copy time and bandwidth
- Use cudaEvent_t to measure time

```
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop );
cudaEventRecord( start, 0 );

kernel<<<blocks,threads>>>( );

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
float elapsedTime;
cudaEventElapsedTime( &elapsedTime, start, stop );
printf( "Elapsed time : %3.1f [ms]\n", elapsedTime );
```

Bandwidth Benchmark

```
int *a, *dev_a;
cudaHostAlloc( (void**)&a, size * sizeof( *a ), cudaHostAllocDefault );
cudaMalloc( (void**)&dev_a, size * sizeof( *dev_a ) );
cudaMemcpy( dev_a, a, size * sizeof( *a ), cudaMemcpyHostToDevice ); cudaMemcpy( a, dev_a,
size * sizeof( *a ), cudaMemcpyDeviceToHost );
```

```
Time using cudaMalloc: 3509.8 ms
 MB/s during copy up: 7293.8
Time using cudaMalloc: 4153.4 ms
 MB/s during copy down: 6163.6
Time using cudaHostAlloc: 2145.9 ms
 MB/s during copy up: 11929.9
Time using cudaHostAlloc: 2097.9 ms
 MB/s during copy down: 12202.9
```

- Note: *cudaHostAllocDefault* parameter

Zero-Copy Host Memory

- Special kind of pinned memory
- Has the all properties of pinned memory
- Use parameter: *cudaHostAllocMapped*
- Additional properties
  - This special host memory can be accessed directly from the device (*zero-copy* memory)
- Example usage: zero-copy dot product

Zero-Copy Host Memory

technische universität dortmund

```
cudaHostAlloc( (void**)&a, size*sizeof(float), cudaHostAllocWriteCombined | cudaHostAllocMapped );
cudaHostAlloc( (void**)&b, size*sizeof(float), cudaHostAllocWriteCombined | cudaHostAllocMapped );
cudaHostAlloc( (void**)&c, blocksPerGrid*sizeof(float), cudaHostAllocMapped );

cudaHostGetDevicePointer( &dev_a, a, 0 );
cudaHostGetDevicePointer( &dev_b, b, 0 );
cudaHostGetDevicePointer( &dev_c,c, 0 );

for (int i=0; i<size; i++) {
 a[i] = i; b[i] = i*2;
}
kernel<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b, dev_c );
cudaThreadsSynchronize();

for (int i=0; i<blocksPerGrid; i++) {
  printf( „c[%i] = %f \n", i, c[i] );
}
```

Zero-Copy Host Memory

- *cudaHostAllocMapped:*
  - Can access memory from GPU
- GPU has different memory space:
  - Buffers have different addresses
- *cudaHostGetDevicePointer*()
  - Get a valid device address for the memory

Streams

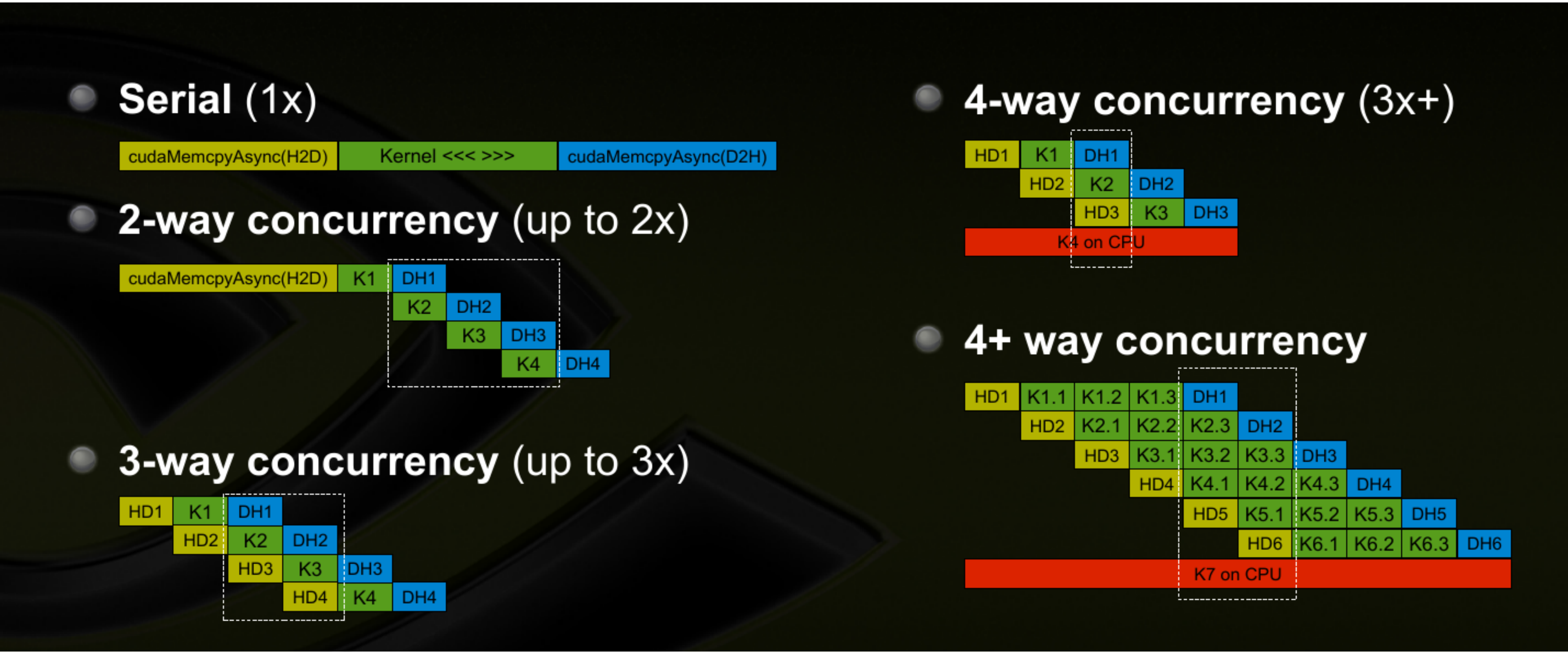
- GPU queue of operations (cudaStream_t)
- Can add kernel launches, memory copies, etc.
- Queue will be executed in order that elements are placed into the queue
- Tasks in streams can execute in parallel
- *Device overlap*: memory copy while performing a kernel calculation

Streams

- Beyond threaded parallelism
- Perform simultaneously:
  - Kernel<<<,>>>
  - cudaMemcpyAsync(H2D)(Pinned Memory)
  - cudaMemcpyAsync(D2H)
  - Operations on the CPU
- Multiple Streams concurrency model:
  - Operations may run concurrently
  - Operations may run interleaved

Streams

Image: Courtesy NVIDIA Coorp.



Streams


```
// allocate pinned memory
cudaMallocHost((void**)&a, size);
cudaMalloc((void**)&dev_a, size);

 // copy the locked memory to the device, async
cudaMemcpyAsync( dev_a, a, size,
 cudaMemcpyHostToDevice, streamid );

kernel<<<blocks,threads,0,streamid>>>( dev_a );

// copy the data from device to locked memory
cudaMemcpyAsync( a, dev_a, size,
 cudaMemcpyDeviceToHost, streamid );

myCPUfunction();
```

Streams

basic_single_stream.cu

```
for (int i=0; i<FULL_DATA_SIZE; i++) {
 host_a[i] = rand();
 host_b[i] = rand();
}
checkCudaErrors( cudaEventRecord( start, 0 ) );
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
 // copy the locked memory to the device, async
 checkCudaErrors( cudaMemcpyAsync( dev_a, host_a+i, N * sizeof(int),
 cudaMemcpyHostToDevice, stream ) );

 checkCudaErrors( cudaMemcpyAsync( dev_b, host_b+i, N * sizeof(int),
 cudaMemcpyHostToDevice, stream ) );
 kernel<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );
 // copy the data from device to locked memory
 checkCudaErrors( cudaMemcpyAsync( host_c+i, dev_c, N * sizeof(int),
 cudaMemcpyDeviceToHost, stream ) );
}
 // copy result chunk from locked to full buffer
 checkCudaErrors(  cudaStreamSynchronize( stream ) );
```

Time taken: 62 ms, nvidia

Time taken: 51.8 ms, 780gtx

Time taken: 24 ms, 980 GTX Ti

- Synchronous function
  - On function return the copy is finished
- Asynchronous function
  - Copy is finished before the next operation in the stream is executed

Multiple Streams

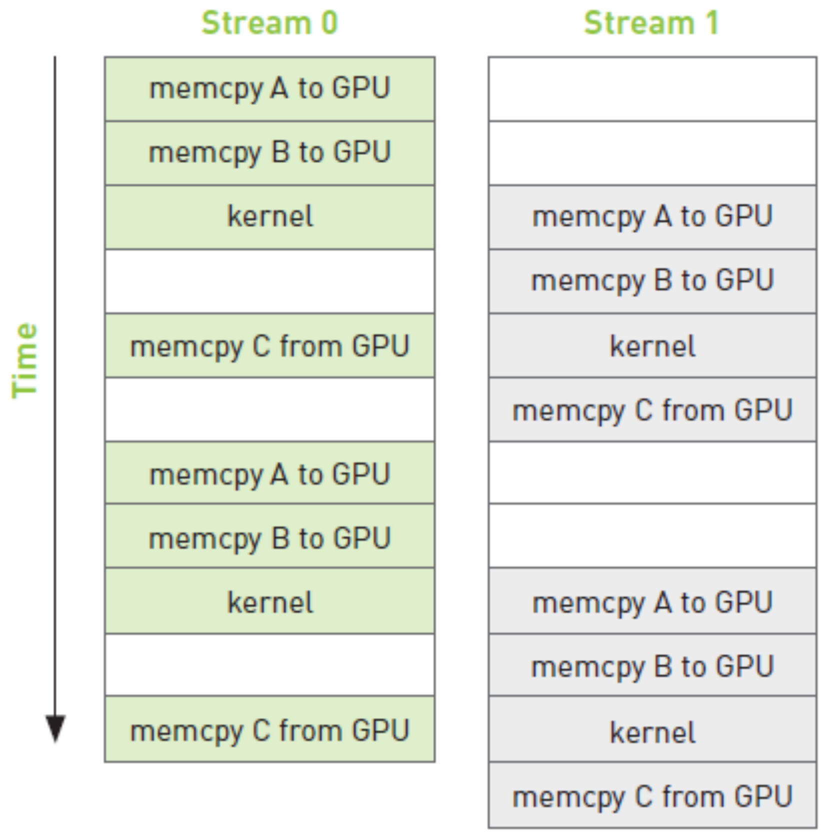


Image: Courtesy NVIDIA Coorp.

- Use a second stream
- Overlap copy with kernel

Multiple Streams

```
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
// copy the locked memory to the device, async
checkCudaErrors( cudaMemcpyAsync( dev_a0, host_a+i, N *
sizeof(int), cudaMemcpyHostToDevice, stream0 ) );

checkCudaErrors( cudaMemcpyAsync( dev_b0, host_b+i, N *
sizeof(int), cudaMemcpyHostToDevice, stream0 ) );

kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
// copy the data from device to locked memory

checkCudaErrors( cudaMemcpyAsync( host_c+i, dev_c0, N *
sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );
 // copy the locked memory to the device, async
checkCudaErrors( cudaMemcpyAsync( dev_a1, host_a+i+N, N *
sizeof(int), cudaMemcpyHostToDevice, stream1 ) );

checkCudaErrors( cudaMemcpyAsync( dev_b1, host_b+i+N, N *
sizeof(int), cudaMemcpyHostToDevice, stream1 ) );

kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
 // copy the data from device to locked memory
checkCudaErrors( cudaMemcpyAsync( host_c+i+N, dev_c1, N *
sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
}
```

Scheduling and Toolkit version differences

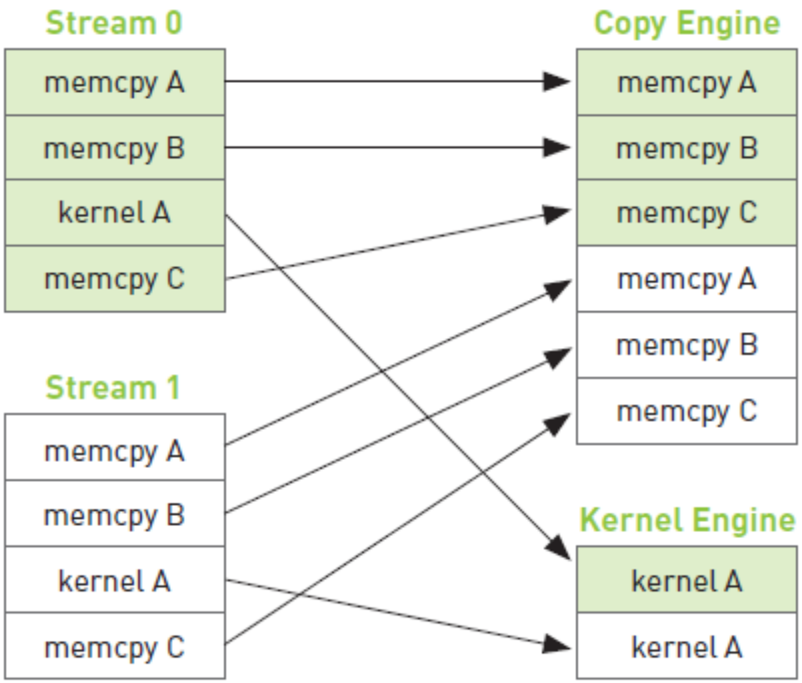
Time taken: 56 ms, 780 gtx

Time taken: 61 ms, nvidia

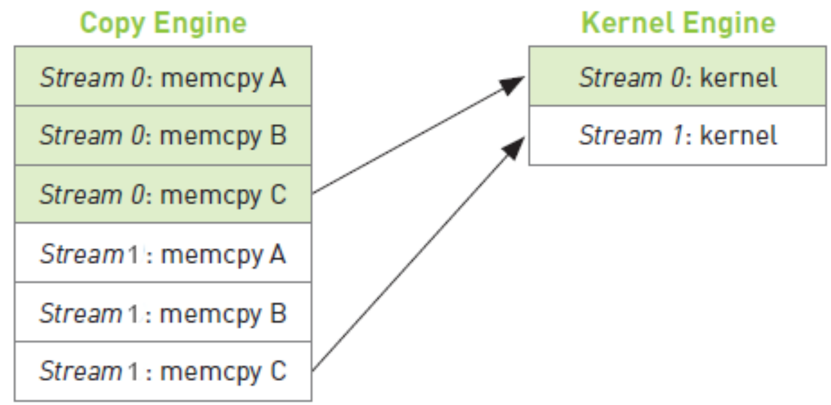
Time taken: 16.4 ms, 980 GTX Ti

Multiple Streams

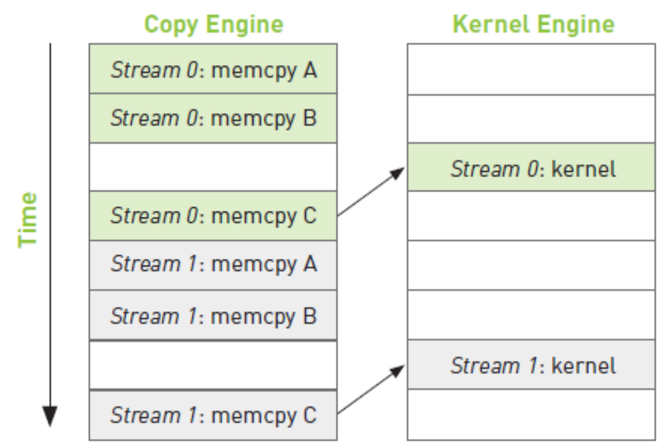
Stream to hardware mapping



Dependencies



Final Scheduling

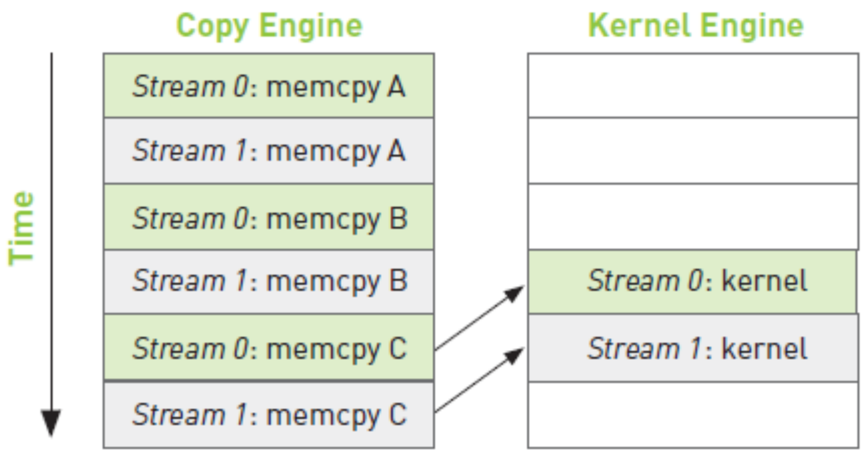


Images: Courtesy NVIDIA Coorp.

Multiple Streams

technische universität dortmund

```
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
 // enqueue copies of a in stream0 and stream1

checkCudaErrors( cudaMemcpyAsync( dev_a0, host_a+i,
N * sizeof(int), cudaMemcpyHostToDevice,
stream0 ) );

checkCudaErrors( cudaMemcpyAsync( dev_a1, host_a+i
+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1
) );

// enqueue copies of b in stream0 and stream1
checkCudaErrors( cudaMemcpyAsync( dev_b0, host_b+i,
N * sizeof(int), cudaMemcpyHostToDevice,
stream0 ) );
checkCudaErrors( cudaMemcpyAsync( dev_b1, host_b+i
+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1
) );

// enqueue kernels in stream0 and stream1
kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0,
dev_c0 ); kernel<<<N/256,256,0,stream1>>>( dev_a1,
dev_b1, dev_c1 );

// enqueue copies of c from device  checkCudaErrors(
cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int),
cudaMemcpyDeviceToHost, stream0 ) );

checkCudaErrors( cudaMemcpyAsync( host_c+i+N,
dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost,
stream1 ) );
}
```



Time taken: 46.5 ms, 10% 780 gtx

Time taken: 48 ms, 21% nvidia

Time taken: 16.4 ms, 33% 980 GTX Ti

CUDA Libraries Overview

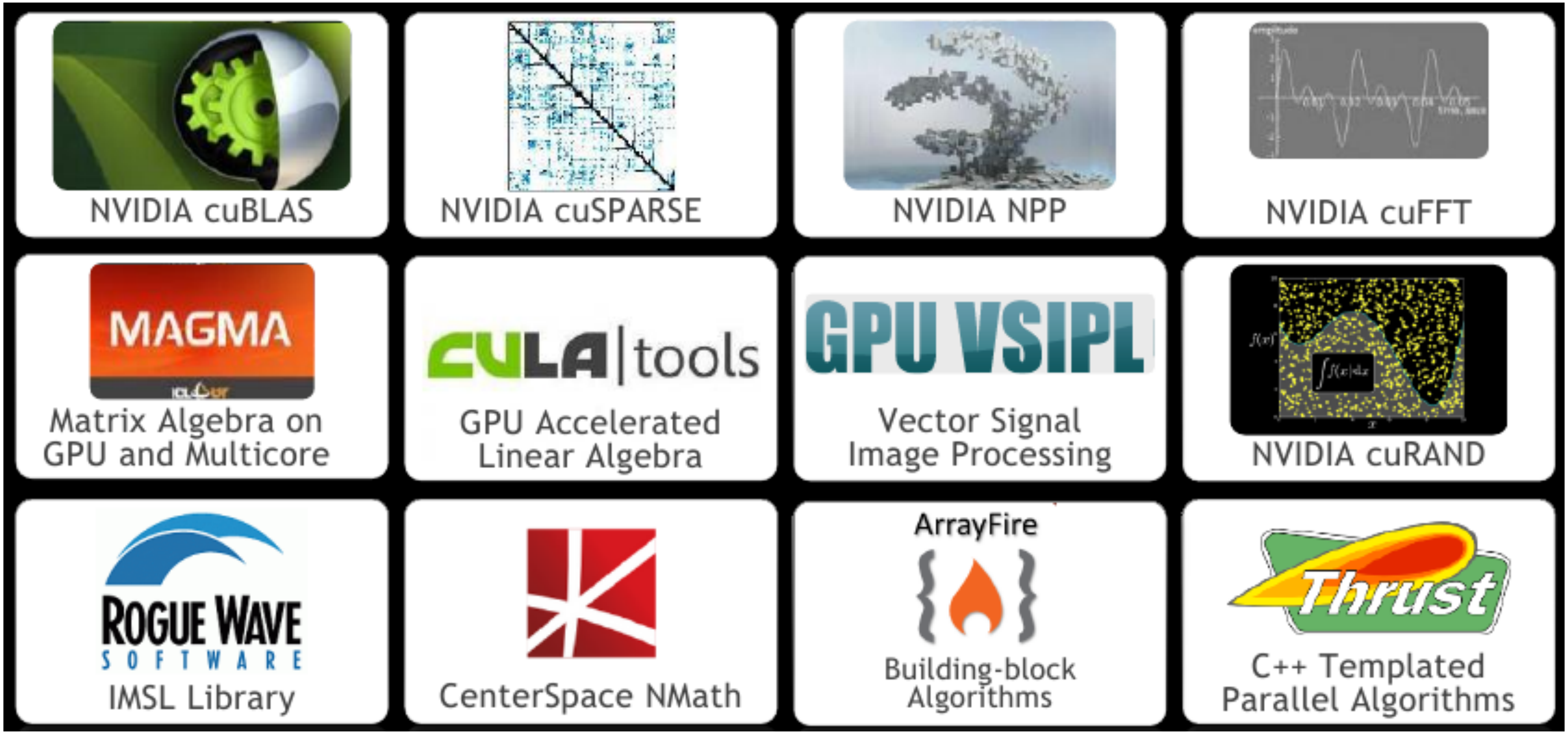


Image: Courtesy NVIDIA Coorp.

Basic Linear Algebra Subprograms

- vector-vector operations
- matrix-vector operations
- matrix-matrix operations
- Uses column-major order as in Fortran
- cuBLAS aims for compatibilty to BLAS and Fortran

- Include <cublas_v2.h>
- Thread-safe
- Works well on multi-GPU systems

cuBLAS by example

- https://developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf

High-level C++ Template Library

- Host and Device containers in STL-Style
- Enhances productivity
- Enhances portability

Flexible

- Backends for CUDA, OpenMP, TBB
- Open source (extension, customization)
- Integrates easily with existing code

Thrust example

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>
#include <iostream>
int main(void) {
 // initialize all ten integers of a device_vector to 1
 thrust::device_vector<int> D(10, 1);
 // set the first seven elements of a vector to 9
 thrust::fill(D.begin(), D.begin() + 7, 9);
 // initialize a host_vector with the first five elements of D
 thrust::host_vector<int> H(D.begin(), D.begin() + 5);
 // set the elements of H to 0, 1, 2, 3, ...
 thrust::sequence(H.begin(), H.end());
 // copy all of H back to the beginning of D
 thrust::copy(H.begin(), H.end(), D.begin());
 // print D
 for(int i = 0; i < D.size(); i++)
   std::cout << "D[" << i << "] = " << D[i] << std::endl;

 return 0; }
```