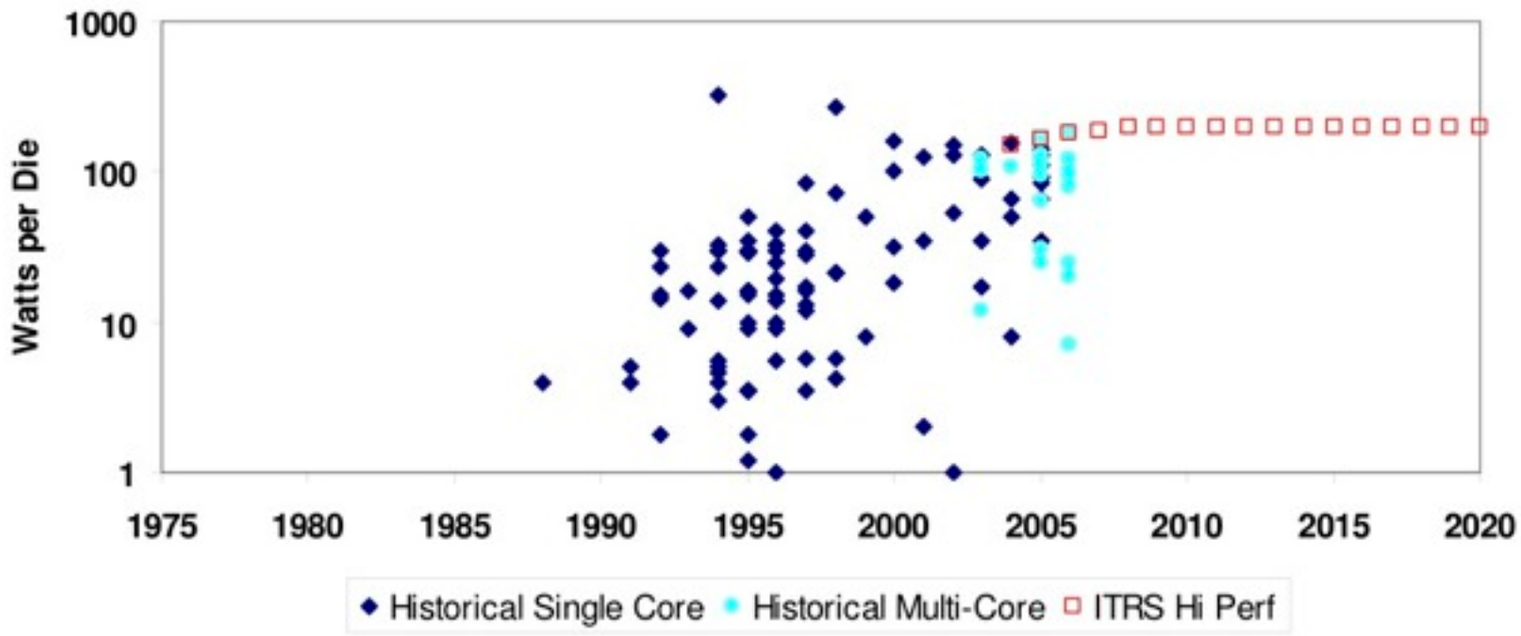
OpenCL for Starters

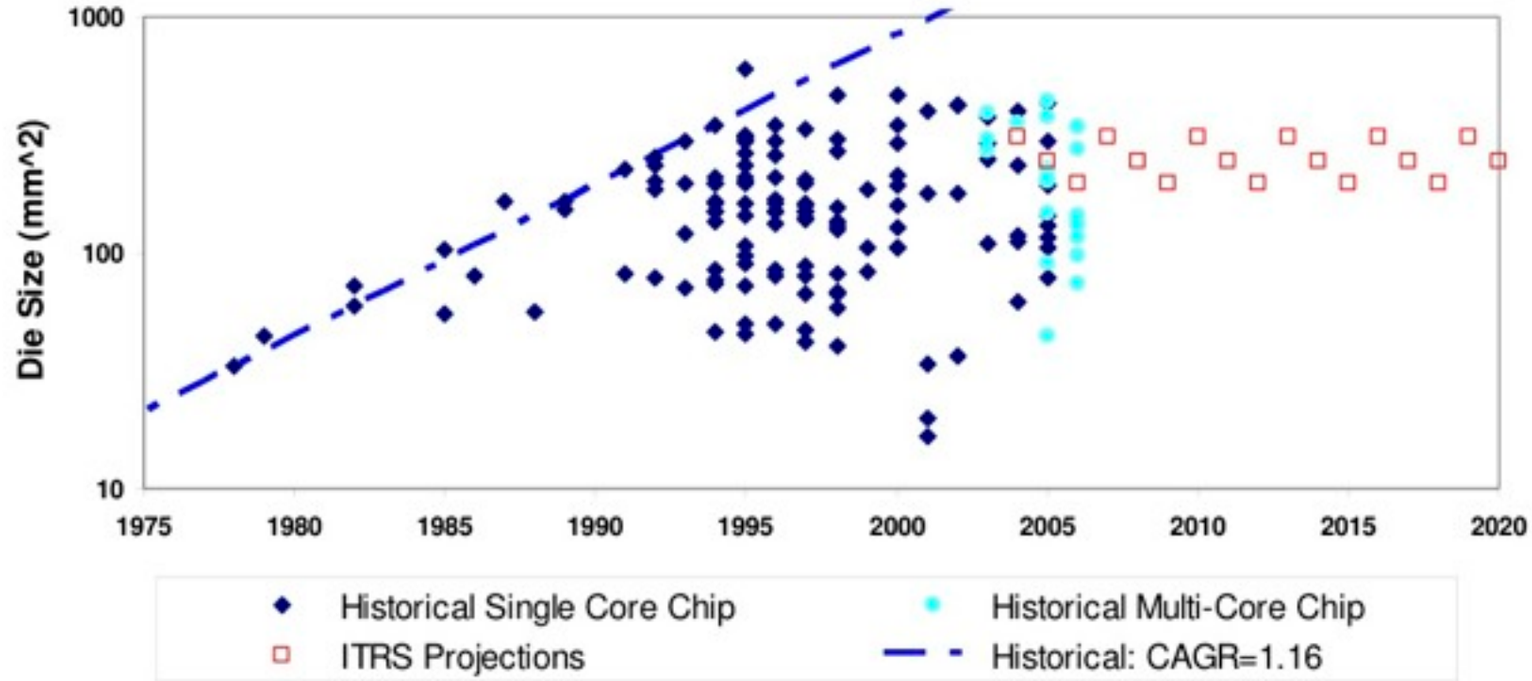
Guillermo Marcus
ZITI - University of Heidelberg
guillermo.marcus@ziti.uni-heidelberg.de

RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

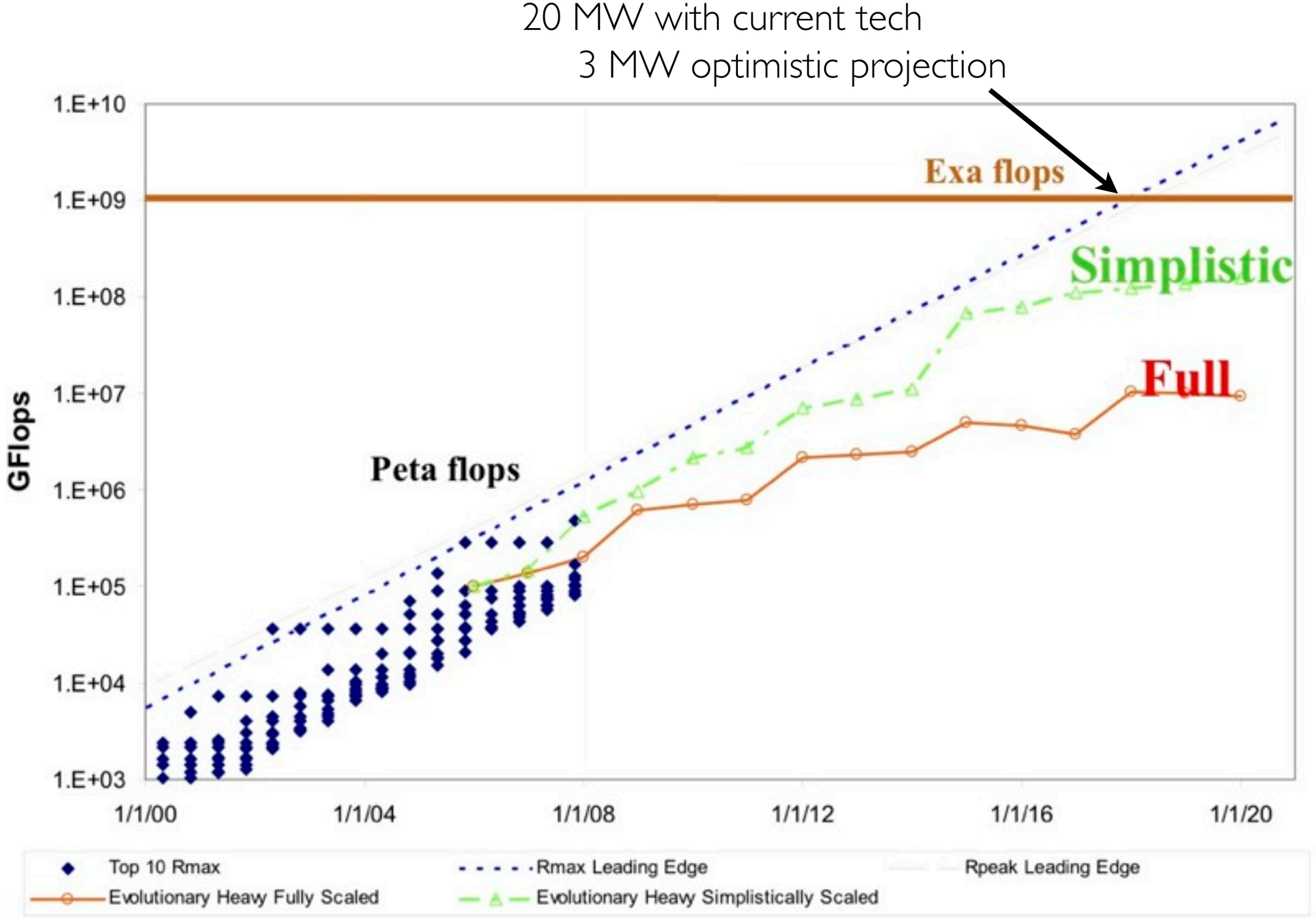
Overview

- Basic Concepts
  - Application Model
  - Memory Model
  - Execution Model
- OpenCL concepts - HelloVector
- OpenCL Events
- NBody in OpenCL





Exascale Computing Study - Final Report. September 2008. DARPA



Exascale Software Study - Final Report. September 2009. DARPA. From slides by Simon Horst (LLNL)

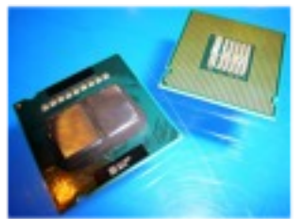
To summarize:

- GPUs are cheap and available in most current systems
- GPUs provide an accelerator platform to complement the CPU performance gap
- Accelerators are needed in the HPC if we want to achieve 1 Exaflop by 2018

Hardware Options

CPUs

Central Processing Units



General Purpose oriented

1-12 Cores

Up to 4 pipes per core using Vector Units

Fully Programmable, many languages available

Very well studied

Max. 125W per processor

GPUs

Graphic Processing Units

Graphics oriented

16-512 Cores

Massively Parallel Architecture, specialized instructions for parallel processing

Fully programmable, but limited languages

Algorithms not fully explored

Max. 400W per card

FPGAs

Field Programmable Gate Arrays

Custom designs, best for processing streaming data

Programmable Logic, Architecture is custom-built for the required application

Requires extensive knowledge to program, development time is longer than CPUs and GPUs

Application interface is custom built on each case

Max. 60W per FPGA

ASICs

Application Specific Integrated Circuits

Fully custom designs, built for a specific application

Not flexible, cannot be changed once it is built

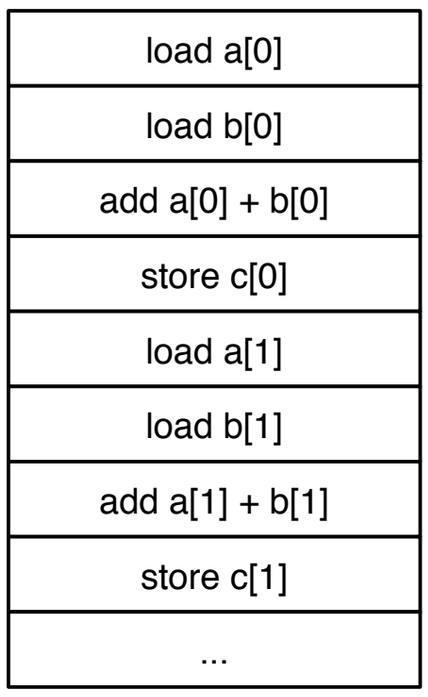
Development is even more specialized than FPGAs

Power consumption varies with the application, usually best performance per Watt

SISD

- Single Instruction, Single Data
- Completely serial execution: one operation after another
- One worker with a single task

```
int a[N], b[N], c[N];
int i;

for(i=0; i<N; i++)
    c[i] = a[i] + b[i];
```

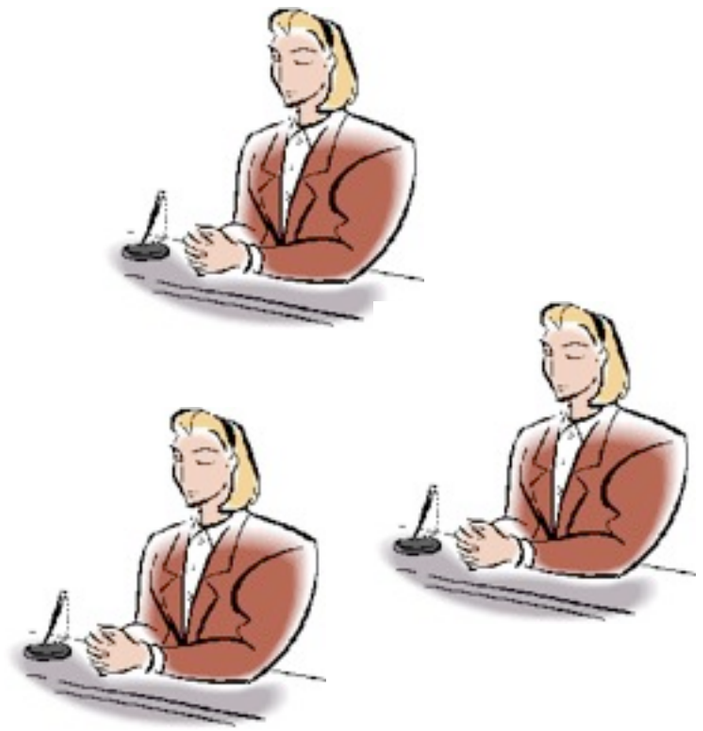


single-thread

single-core

MIMD

- Multiple Instruction, Multiple Data
- Every instruction thread is independent and works over an independent data stream
- Every worker can do completely unrelated tasks



worker 1

```
int a[N], b[N], c[N];
int i;

for(i=0; i<N; i++)
     c[i] = a[i] + b[i];
```

worker 2

```
int x[N], y[N];
int j;

for(j=0; j<N; j++)
     y[i] = 4*(x*x) - 3*x + 1;
```

multi-thread

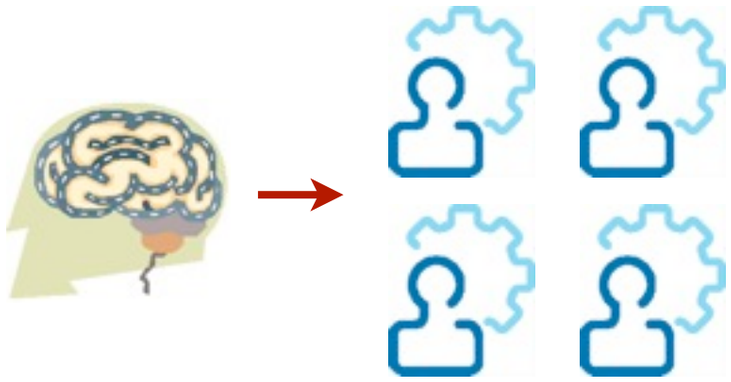
multi-core

clusters

nodes

SIMD

- Single Instruction, Multiple Data
- One stream of instructions is applied to multiple streams of data
- Multiple workers, but only one task
- Parallelism is limited by the number of workers



```
int a[N], b[N], c[N];
int i;

for(i=0; i<N; i+=4)
     add4( c[i], a[i], b[i] );
```

load4 a[0]
load4 b[0]
add4 a[0] + b[0]
store4 c[0]
load4 a[4]
load4 b[4]
add4 a[4] + b[4]
store4 c[4]
...

worker 1	worker 2	worker 3	worker 4
a[0]	a[1]	a[2]	a[3]
b[0]	b[1]	b[2]	b[3]
c[0]	c[1]	c[2]	c[3]
st c[0]	st c[1]	st c[2]	st c[3]
a[4]	a[5]	a[6]	a[7]
b[4]	b[5]	b[6]	b[7]
c[4]	c[5]	c[6]	c[7]
st c[4]	st c[5]	st c[6]	st c[7]

Vector Units

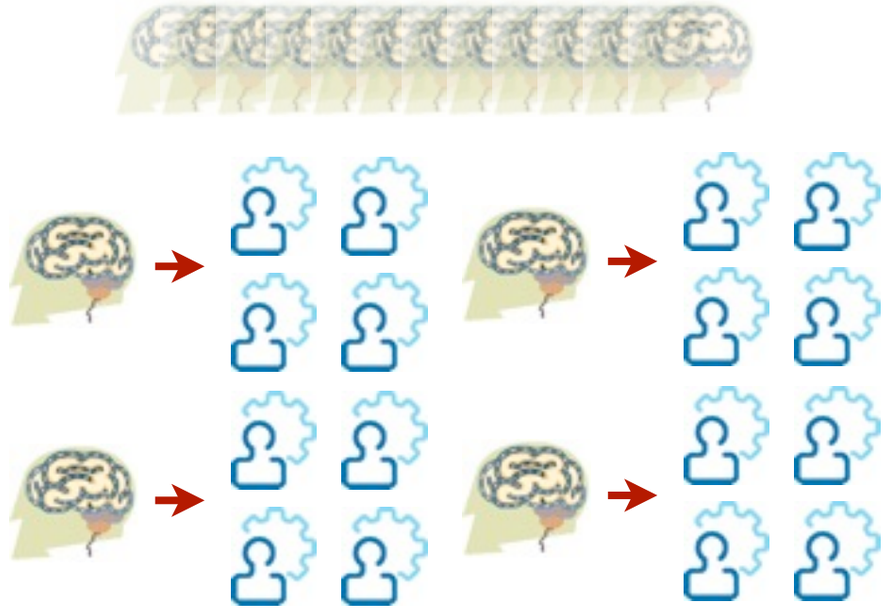
SSE

Altivec

AVX

SIMT

- Single Instruction, Multiple Threads
- Combines the flexibility of threads with the efficiency of SIMD
- Normally, there are many more threads than workers



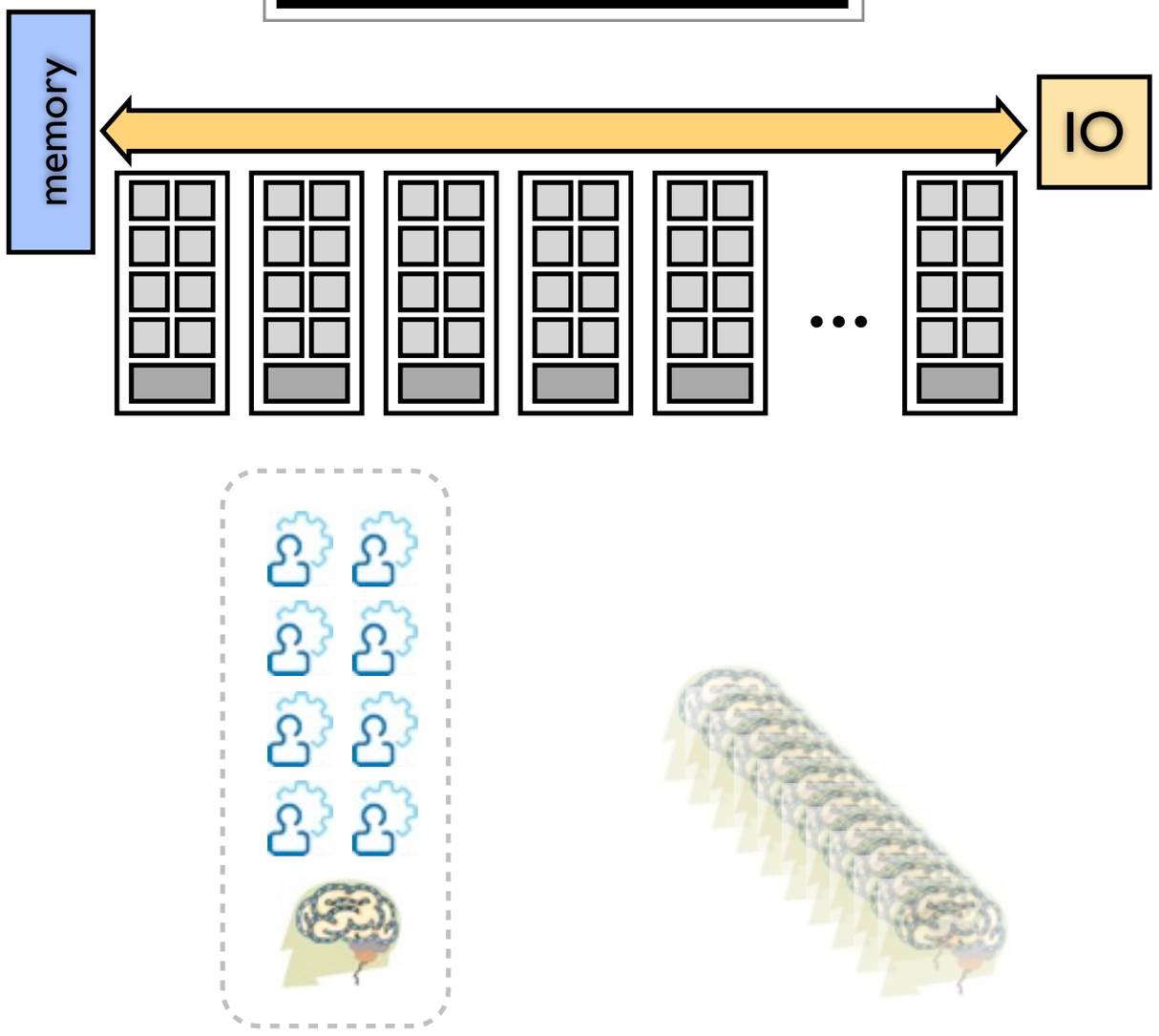
```
int a[N], b[N], c[N];
int tid;

tid = getThreadID();
c[tid] = a[tid] + b[tid];
```

load4 a[0]
load4 b[0]
add4 a[0] + b[0]
store4 c[0]
load4 a[4]
load4 b[4]
add4 a[4] + b[4]
store4 c[4]
...

worker 1	worker 2	worker 3	worker 4
a[0]	a[1]	a[2]	a[3]
b[0]	b[1]	b[2]	b[3]
c[0]	c[1]	c[2]	c[3]
st c[0]	st c[1]	st c[2]	st c[3]
a[4]	a[5]	a[6]	a[7]
b[4]	b[5]	b[6]	b[7]
c[4]	c[5]	c[6]	c[7]
st c[4]	st c[5]	st c[6]	st c[7]

What is a GPU?

- GPU: Graphics Processing Units
- normally used for high end graphics in computers
- Modern GPUs are programmable and can be used with compute tasks
- They may have many cores (hundreds to thousands)
- A core is equivalent to a worker
- cores are grouped in "Symmetric Multiprocessors"

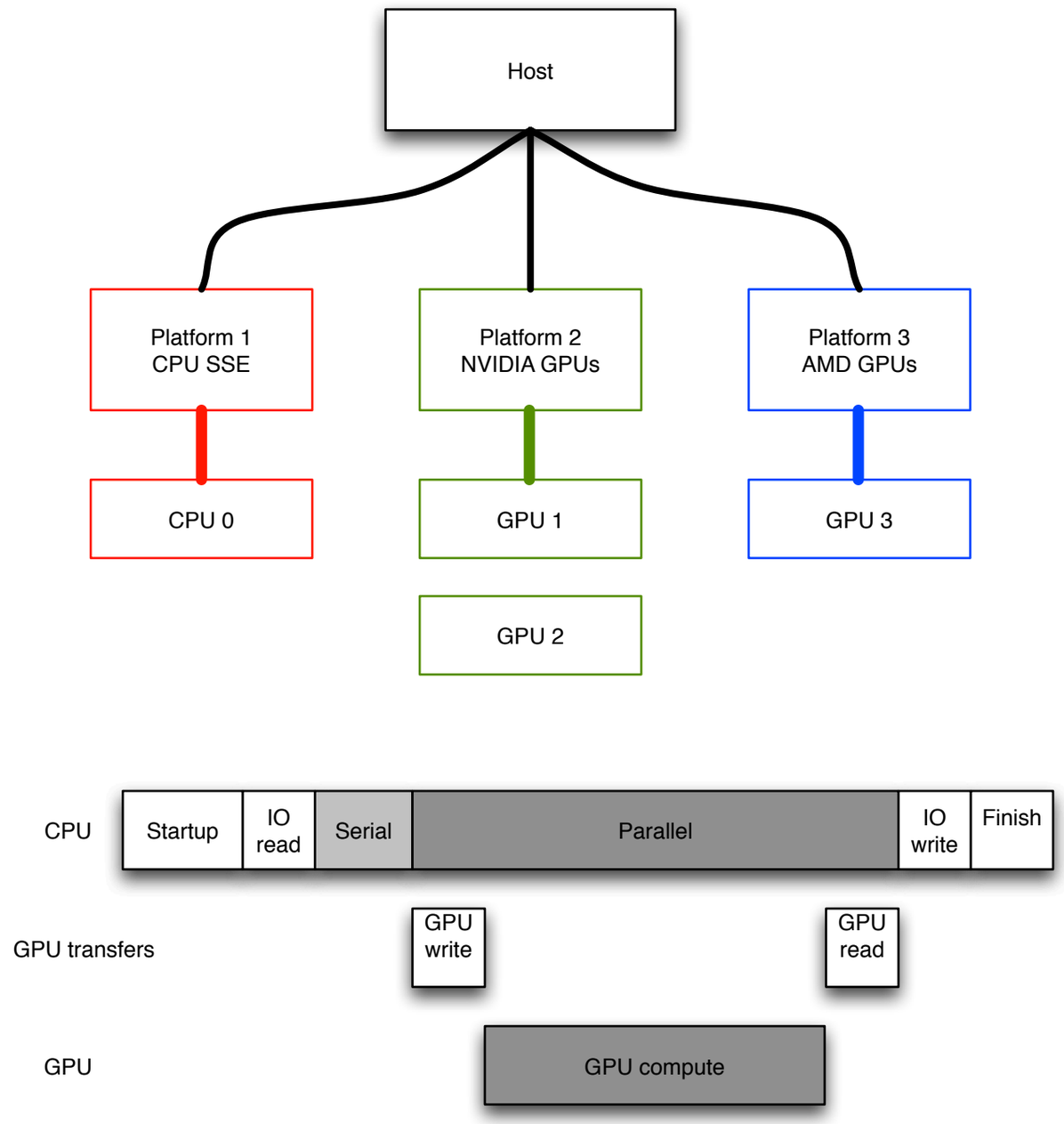


What is OpenCL?

- An Open standard for Computing
  - CL stands for Computing Language
- Handled by the Khronos group www.khronos.org
- Created by joint work of Intel, NVIDIA, AMD and Apple
- It is not limited to a single platform
  - Mostly platform independent*
  - Code is portable, optimizations are NOT !!

Anatomy of a GPU Application

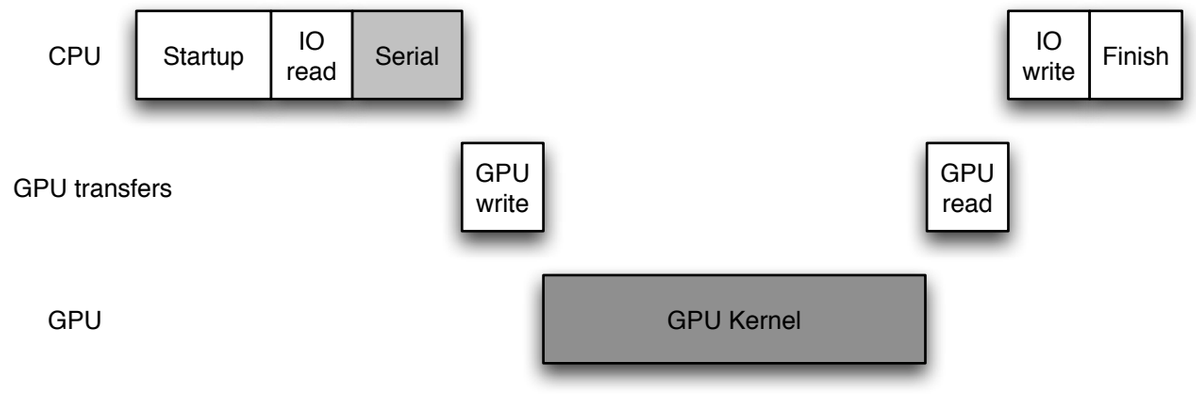
- Activities are driven by the Host program
- We can have
  - Multiple platforms
  - Multiple devices per platform
- The Host application:
  - Select the device(s) to use
  - Directs data transfers between the host and GPU
  - Defines the GPU code to execute



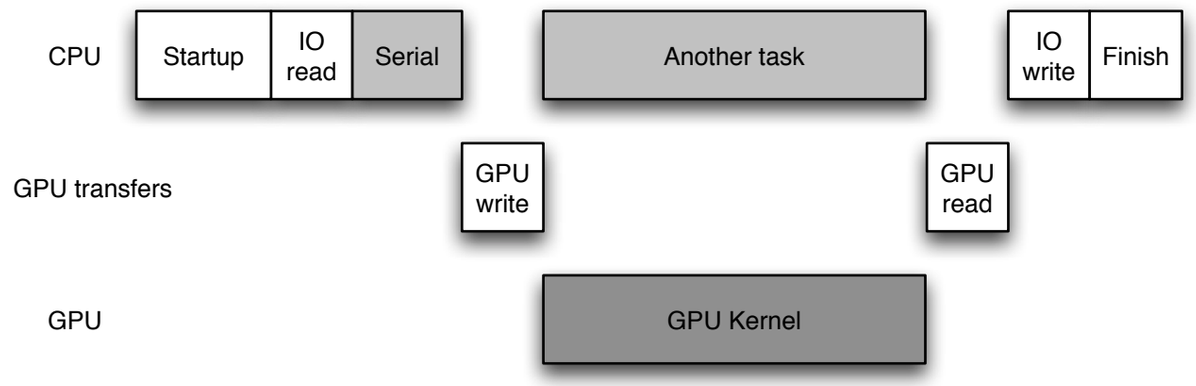
GPU Kernel

- Starts a computation in the GPU
- "Launches"(starts) a collection of threads
- Can be blocking or non-blocking
- Most GPUs can run only one kernel at the same time

Blocking

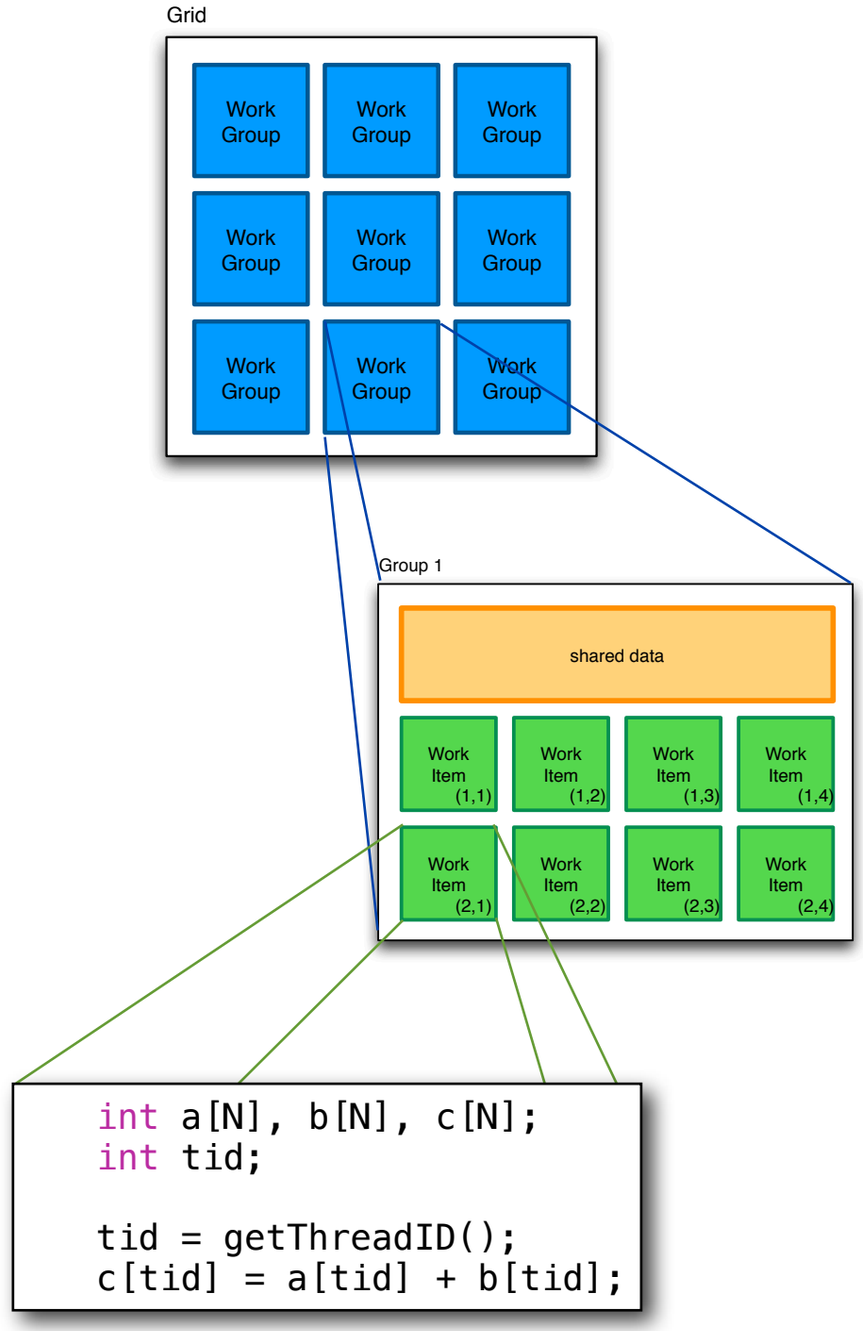


Non-Blocking



Workers and Thread Hierarchy

- A GPU has a fixed number of working units
- A GPU Kernel wants to execute a number of threads
  - There are normally much more threads than workers
  - Every thread executes the same program (as they are SIMT)
- Threads are therefore bundled and divided in independent work groups
- The groups are then scheduled by the GPU to execute in the available working units
- The organization of this thread hierarchy is the kernel specification, and the code executed by each thread is the kernel code. Both are required to define a GPU kernel!



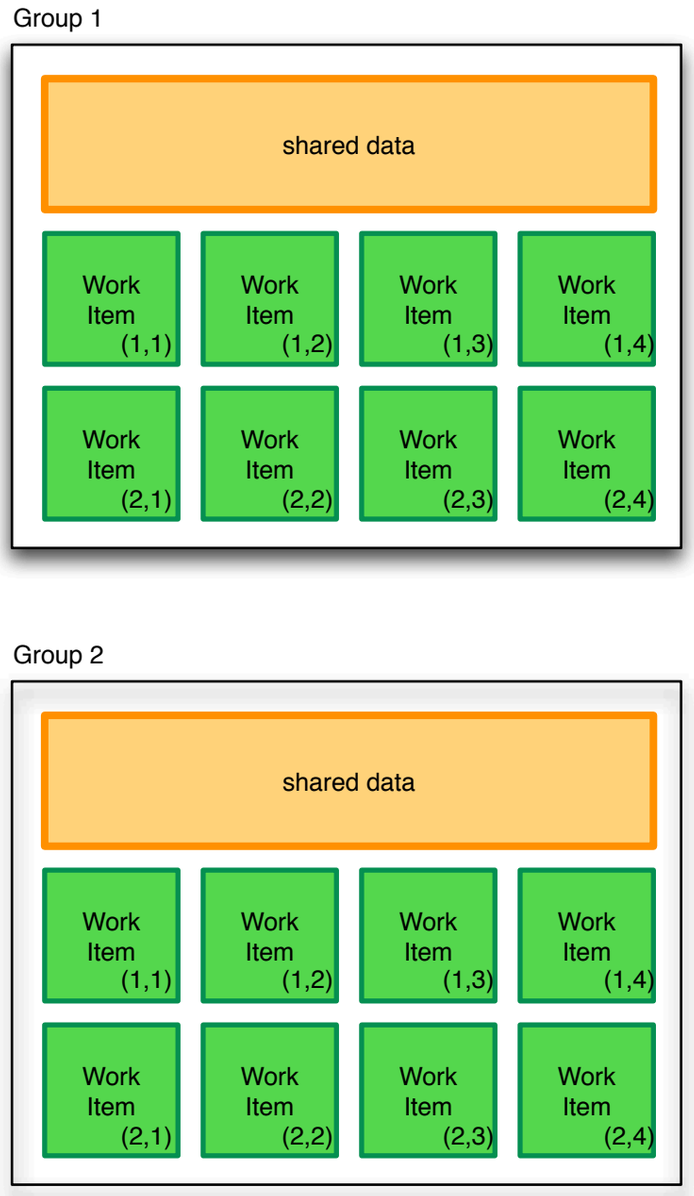
Work Items

- Represents a single thread in the GPU
- Every thread shares the same code
- Every item has a unique (global) ID
- Can have local (private) data that belong to that thread only

```
int a[N], b[N], c[N];
int tid;

tid = getThreadID();
c[tid] = a[tid] + b[tid];
```

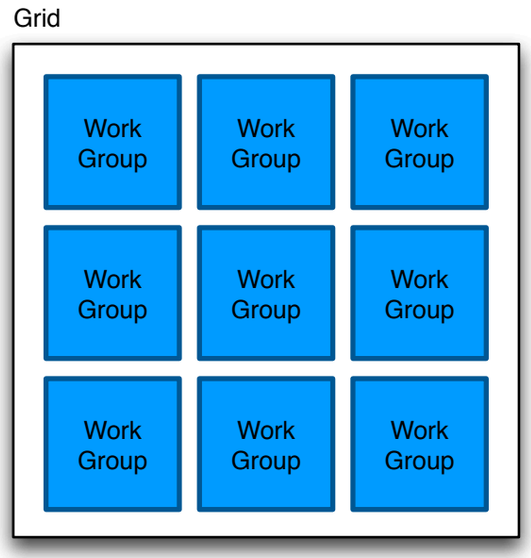
Work Groups

- Work Groups are collections of Work Items
- Items inside a group are executed in parallel
- Items inside a group can share local data
- Groups are independent from each other
- Items can be organized as 1D, 2D or 3D arrays
- Every group has an unique id
- Every item has a local id inside its own group



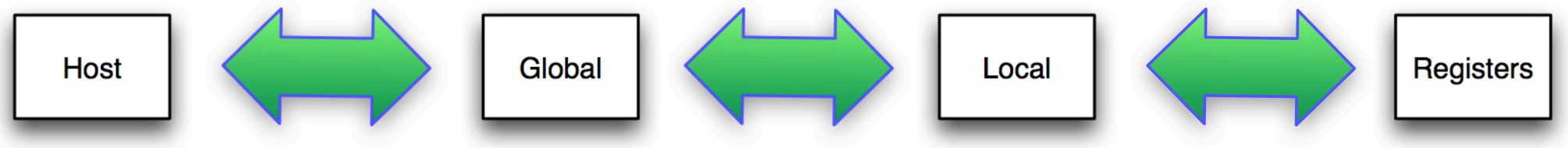
Work Group Grid

- Work Groups are organized in a grid (1D, 2D or 3D) as part of the Kernel definition
- No communications between groups
- No synchronization between groups*
- Grids allow the organization of the groups in a way suitable for the problem



Memory Model

- Hierarchy: Host, Global GPU, Local GPU, Registers
- Moving data between levels is expensive
- Every area has its own constraints



Memory Overview

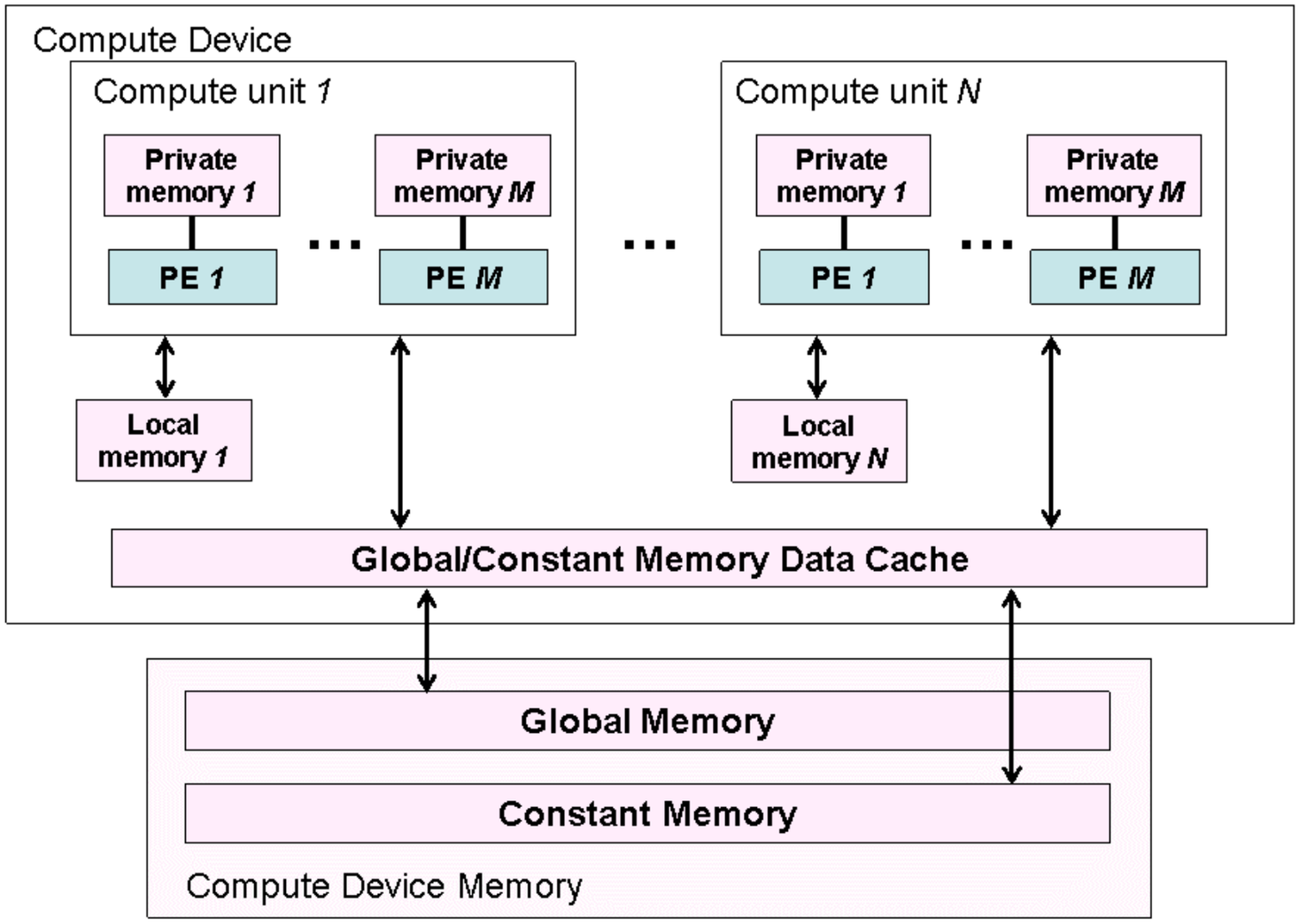


Figure 3.3: *Conceptual OpenCL device architecture with processing elements (PE), compute units and devices. The host is not shown.*

From the OpenCL Specification

Host Memory

- Main Memory of the Host
- Move data between host and main GPU memory
- Transfer is always initiated by the host application
- Can by synchronous or asynchronous
- Bandwidth is limited by the PCIe links

GPU Global Memory

- Main GPU memory, available by all work items
- Biggest in size, up to several GBs
- Huge bandwidth, but also huge latency
  - bandwidth is distributed among SMs
  - Latency is ~400-800 cycles
- Non-cached, except:
  - Textures
  - Newer architectures
- Performance is very dependent in access patterns

GPU Local Memory

- Available to all work items inside a work group
- Limited in size: a few Kilobytes (8-32KB)
- Latency is very similar to registers, therefore latency is very low
- In some platforms, access is constraint by several rules: forces to use several access patterns and avoid others
- Used mostly as scratchpad to reduce global memory access

GPU Registers

- Private to every work item
- Normally hidden, optimized by the compiler
- Fastest access
- Some platforms require registers for the operations (i.e. Fermi); others can also operate in local memory

Constant Memory

- Read only memory
- Cached

- Good for storing Look-Up Tables and non-changeable values
- It is normally a small area of the global memory

Private Memory

- Unique to every work item
- Normally mapped to global memory, as needed

Memory access and allocation

Table 3.1 describes whether the kernel or the host can allocate from a memory region, the type of allocation (static i.e. compile time vs dynamic i.e. runtime) and the type of access allowed i.e. whether the kernel or the host can read and/or write to a memory region.

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation Read / Write access	Dynamic allocation Read / Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read / Write access	Static allocation Read-only access	Static allocation Read / Write access	Static allocation Read / Write access

Table 3.1 *Memory Region - Allocation and Memory Access Capabilites*

From the OpenCL Specification

Memory Overview

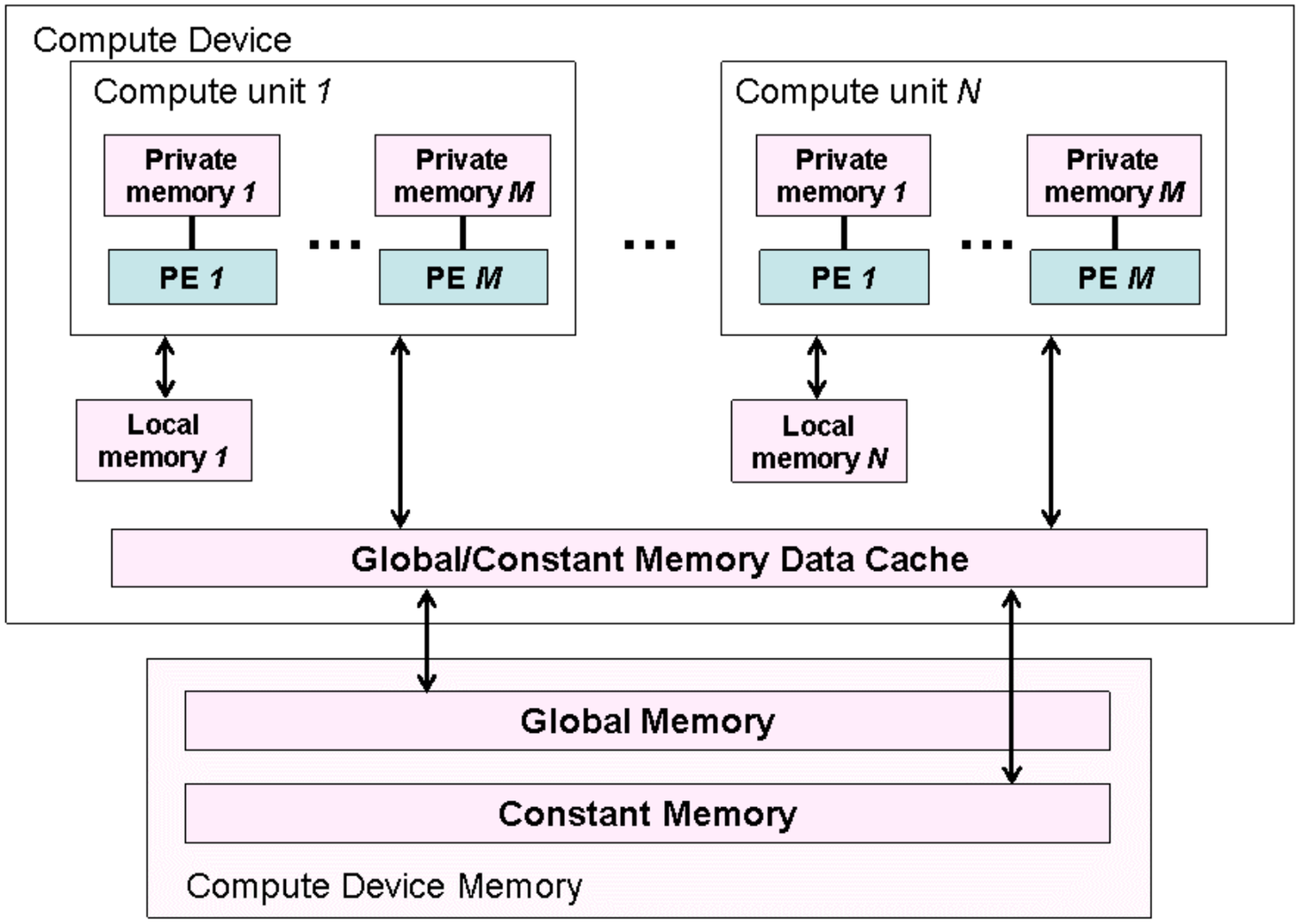
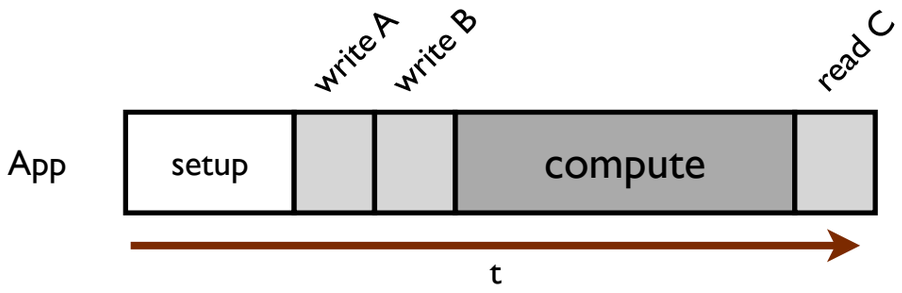


Figure 3.3: *Conceptual OpenCL device architecture with processing elements (PE), compute units and devices. The host is not shown.*

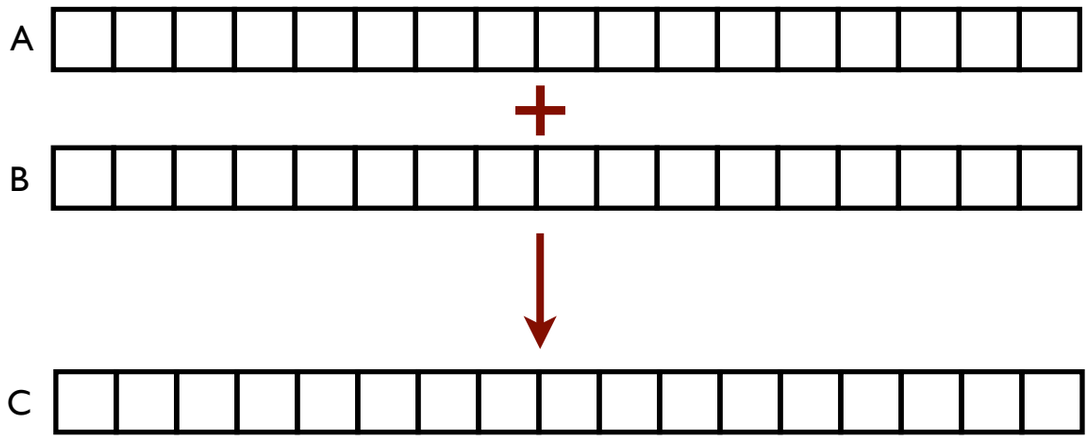
From the OpenCL Specification

Example 1: Vector Addition



- Every work item computes a single element of the result vector C
- Every work item is independent: does not require shared data with any other item
- We need to group the items for the computation: Vectors are linear, so we can organize the grid and the groups as segments of it, as long as the number of threads is N (the size of the vectors)

N threads



$$\vec{C} = \vec{A} + \vec{B}$$

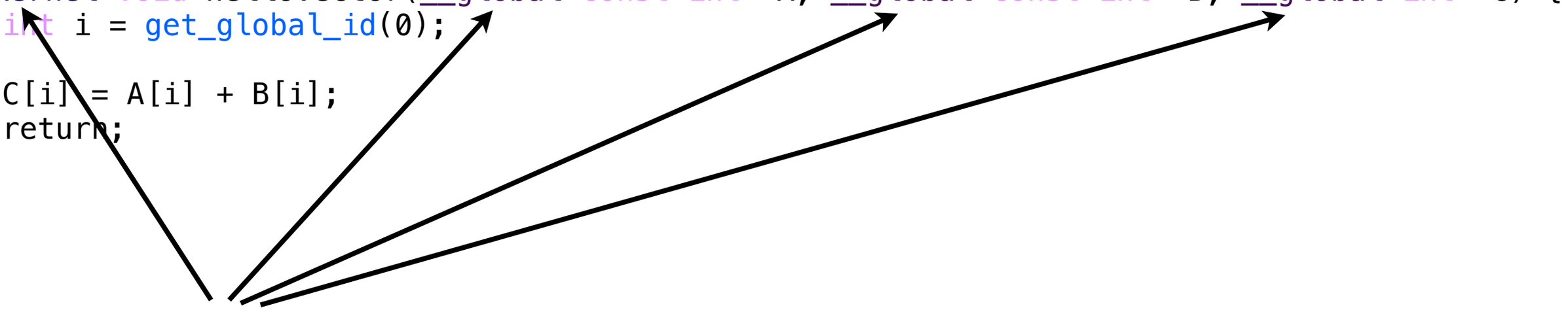
```
int a[N], b[N], c[N];
int tid;

tid = getThreadID();
c[tid] = a[tid] + b[tid];
```

OpenCL Kernel

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

OpenCL Kernel

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```



OpenCL directives

OpenCL Kernel

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

OpenCL Global Thread ID

OpenCL Kernel

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

Vector Element addition

OpenCL Kernel

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

Main Program

```
/**
 * This is the Main file for the OpenCL Hello Vector Program.
 *
 * \author: Guillermo Marcus
 * \date: 2011-11-09
 */

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
	const int VECT_SIZE = 100000;
	int *A = new int[VECT_SIZE];
	int *B = new int[VECT_SIZE];
	int *C = new int[VECT_SIZE];
	bool error = false;

	// Fill test data
	for( int i=0; i < VECT_SIZE; i++ ) {
		A[i] = i;
		B[i] = VECT_SIZE - i;
		C[i] = 0;
	}

	try {
		// Check available platforms
		std::vector<Platform> platforms;
		Platform::get( &platforms );

		// Create context
		cl_context_properties cps[3] = {
			CL_CONTEXT_PLATFORM,
			(cl_context_properties)(platforms[0])(),
			0
		};
		Context context( CL_DEVICE_TYPE_GPU, cps );

		// Devices available in this platform
		std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

		// query name of device
		STRING_CLASS name;
		devices[0].getInfo( CL_DEVICE_NAME, &name );
		cout << "Will use this device: " << name << endl;

		// Read kernel source and build a program
		std::string kernel_source = loadProgram("oclHelloVector.ocl");

		Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

		Program program = Program(context, source);
		program.build(devices);

		// Create a kernel
		Kernel kernel(program, "helloVector");

		// Create command queue using the first device
		CommandQueue queue = CommandQueue( context, devices[0], 0 );

		// Create Memory buffers
		Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

		// Copy buffers to device
		queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
		queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

		// Create kernel specification (ND range)
		NDRange global(VECT_SIZE);
		NDRange local(1);

		// Set kernel arguments
		kernel.setArg(0, bufA);
		kernel.setArg(1, bufB);
		kernel.setArg(2, bufC);

		// Run kernel
		Event event;
		queue.enqueueNDRangeKernel(   kernel, NullRange, global, local );

		// Copy result buffer from device
		queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

	}
	catch (Error err) {
		cerr << "ERROR: " << err.what()	<< "(" << err.err() << ")" << endl;
		error = true;
	}

	// Compare results
	bool test = true;
	for( int i=0; i < VECT_SIZE; i++ ) {
		if (C[i] != VECT_SIZE)
			test = false;
	}
	if (test)
		cout << "Test PASSED!" << endl;
	else
		cout << "Test FAILED!" << endl;

	delete A;delete B;delete C;
	A = NULL;B = NULL;C = NULL;

	if (error)
		return EXIT_FAILURE;
	else
		return EXIT_SUCCESS;

}

std::string loadProgram(const char *filename) {
	ifstream src(filename);
	std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

	return src_code;
}
```

Main Program

Headers

```
/**
 * This is the Main file for the OpenCL Hello Vector Program.
 *
 * \author: Guillermo Marcus
 * \date: 2011-11-09
 */

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
	const int VECT_SIZE = 100000;
	int *A = new int[VECT_SIZE];
	int *B = new int[VECT_SIZE];
	int *C = new int[VECT_SIZE];
	bool error = false;

	// Fill test data
	for( int i=0; i < VECT_SIZE; i++ ) {
		A[i] = i;
		B[i] = VECT_SIZE - i;
		C[i] = 0;
	}

	try {
		// Check available platforms
		std::vector<Platform> platforms;
		Platform::get( &platforms );

		// Create context
		cl_context_properties cps[3] = {
			CL_CONTEXT_PLATFORM,
			(cl_context_properties)(platforms[0])(),
			0
		};
		Context context( CL_DEVICE_TYPE_GPU, cps );

		// Devices available in this platform
		std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

		// query name of device
		STRING_CLASS name;
		devices[0].getInfo( CL_DEVICE_NAME, &name );
		cout << "Will use this device: " << name << endl;

		// Read kernel source and build a program
		std::string kernel_source = loadProgram("oclHelloVector.ocl");

		Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

		Program program = Program(context, source);
		program.build(devices);

		// Create a kernel
		Kernel kernel(program, "helloVector");

		// Create command queue using the first device
		CommandQueue queue = CommandQueue( context, devices[0], 0 );

		// Create Memory buffers
		Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

		// Copy buffers to device
		queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
		queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

		// Create kernel specification (ND range)
		NDRange global(VECT_SIZE);
		NDRange local(1);

		// Set kernel arguments
		kernel.setArg(0, bufA);
		kernel.setArg(1, bufB);
		kernel.setArg(2, bufC);

		// Run kernel
		Event event;
		queue.enqueueNDRangeKernel(   kernel, NullRange, global, local );

		// Copy result buffer from device
		queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

	}
	catch (Error err) {
		cerr << "ERROR: " << err.what()	<< "(" << err.err() << ")" << endl;
		error = true;
	}

	// Compare results
	bool test = true;
	for( int i=0; i < VECT_SIZE; i++ ) {
		if (C[i] != VECT_SIZE)
			test = false;
	}
	if (test)
		cout << "Test PASSED!" << endl;
	else
		cout << "Test FAILED!" << endl;

	delete A;delete B;delete C;
	A = NULL;B = NULL;C = NULL;

	if (error)
		return EXIT_FAILURE;
	else
		return EXIT_SUCCESS;

}

std::string loadProgram(const char *filename) {
	ifstream src(filename);
	std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

	return src_code;
}
```

Main Program

Setup

```
/**
 * This is the Main file for the OpenCL Hello Vector Program.
 *
 * \author: Guillermo Marcus
 * \date: 2011-11-09
 */

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
	const int VECT_SIZE = 100000;
	int *A = new int[VECT_SIZE];
	int *B = new int[VECT_SIZE];
	int *C = new int[VECT_SIZE];
	bool error = false;

	// Fill test data
	for( int i=0; i < VECT_SIZE; i++ ) {
		A[i] = i;
		B[i] = VECT_SIZE - i;
		C[i] = 0;
	}

	try {
		// Check available platforms
		std::vector<Platform> platforms;
		Platform::get( &platforms );

		// Create context
		cl_context_properties cps[3] = {
			CL_CONTEXT_PLATFORM,
			(cl_context_properties)(platforms[0])(),
			0
		};
		Context context( CL_DEVICE_TYPE_GPU, cps );

		// Devices available in this platform
		std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

		// query name of device
		STRING_CLASS name;
		devices[0].getInfo( CL_DEVICE_NAME, &name );
		cout << "Will use this device: " << name << endl;

		// Read kernel source and build a program
		std::string kernel_source = loadProgram("oclHelloVector.ocl");

		Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

		Program program = Program(context, source);
		program.build(devices);

		// Create a kernel
		Kernel kernel(program, "helloVector");

		// Create command queue using the first device
		CommandQueue queue = CommandQueue( context, devices[0], 0 );

		// Create Memory buffers
		Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

		// Copy buffers to device
		queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
		queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

		// Create kernel specification (ND range)
		NDRange global(VECT_SIZE);
		NDRange local(1);

		// Set kernel arguments
		kernel.setArg(0, bufA);
		kernel.setArg(1, bufB);
		kernel.setArg(2, bufC);

		// Run kernel
		Event event;
		queue.enqueueNDRangeKernel(   kernel, NullRange, global, local );

		// Copy result buffer from device
		queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

	}
	catch (Error err) {
		cerr << "ERROR: " << err.what()	<< "(" << err.err() << ")" << endl;
		error = true;
	}

	// Compare results
	bool test = true;
	for( int i=0; i < VECT_SIZE; i++ ) {
		if (C[i] != VECT_SIZE)
			test = false;
	}
	if (test)
		cout << "Test PASSED!" << endl;
	else
		cout << "Test FAILED!" << endl;

	delete A;delete B;delete C;
	A = NULL;B = NULL;C = NULL;

	if (error)
		return EXIT_FAILURE;
	else
		return EXIT_SUCCESS;

}

std::string loadProgram(const char *filename) {
	ifstream src(filename);
	std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

	return src_code;
}
```

Main Program

OpenCL App

```
/**
 * This is the Main file for the OpenCL Hello Vector Program.
 *
 * \author: Guillermo Marcus
 * \date: 2011-11-09
 */

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
	const int VECT_SIZE = 100000;
	int *A = new int[VECT_SIZE];
	int *B = new int[VECT_SIZE];
	int *C = new int[VECT_SIZE];
	bool error = false;

	// Fill test data
	for( int i=0; i < VECT_SIZE; i++ ) {
		A[i] = i;
		B[i] = VECT_SIZE - i;
		C[i] = 0;
	}

	try {
		// Check available platforms
		std::vector<Platform> platforms;
		Platform::get( &platforms );

		// Create context
		cl_context_properties cps[3] = {
			CL_CONTEXT_PLATFORM,
			(cl_context_properties)(platforms[0])(),
			0
		};
		Context context( CL_DEVICE_TYPE_GPU, cps );

		// Devices available in this platform
		std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

		// query name of device
		STRING_CLASS name;
		devices[0].getInfo( CL_DEVICE_NAME, &name );
		cout << "Will use this device: " << name << endl;

		// Read kernel source and build a program
		std::string kernel_source = loadProgram("oclHelloVector.ocl");

		Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

		Program program = Program(context, source);
		program.build(devices);

		// Create a kernel
		Kernel kernel(program, "helloVector");

		// Create command queue using the first device
		CommandQueue queue = CommandQueue( context, devices[0], 0 );

		// Create Memory buffers
		Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

		// Copy buffers to device
		queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
		queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

		// Create kernel specification (ND range)
		NDRange global(VECT_SIZE);
		NDRange local(1);

		// Set kernel arguments
		kernel.setArg(0, bufA);
		kernel.setArg(1, bufB);
		kernel.setArg(2, bufC);

		// Run kernel
		Event event;
		queue.enqueueNDRangeKernel(   kernel, NullRange, global, local );

		// Copy result buffer from device
		queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

	}
	catch (Error err) {
		cerr << "ERROR: " << err.what()	<< "(" << err.err() << ")" << endl;
		error = true;
	}

	// Compare results
	bool test = true;
	for( int i=0; i < VECT_SIZE; i++ ) {
		if (C[i] != VECT_SIZE)
			test = false;
	}
	if (test)
		cout << "Test PASSED!" << endl;
	else
		cout << "Test FAILED!" << endl;

	delete A;delete B;delete C;
	A = NULL;B = NULL;C = NULL;

	if (error)
		return EXIT_FAILURE;
	else
		return EXIT_SUCCESS;

}

std::string loadProgram(const char *filename) {
	ifstream src(filename);
	std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

	return src_code;
}
```

Main Program

```
/**
 * This is the Main file for the OpenCL Hello Vector Program.
 *
 * \author: Guillermo Marcus
 * \date: 2011-11-09
 */

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
      const int VECT_SIZE = 100000;
      int *A = new int[VECT_SIZE];
      int *B = new int[VECT_SIZE];
      int *C = new int[VECT_SIZE];
      bool error = false;

      // Fill test data
      for( int i=0; i < VECT_SIZE; i++ ) {
               A[i] = i;
               B[i] = VECT_SIZE - i;
               C[i] = 0;
      }

      try {
               // Check available platforms
               std::vector<Platform> platforms;
               Platform::get( &platforms );

               // Create context
               cl_context_properties cps[3] = {
                      CL_CONTEXT_PLATFORM,
                      (cl_context_properties)(platforms[0])(),
                      0
               };
               Context context( CL_DEVICE_TYPE_GPU, cps );

               // Devices available in this platform
               std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

               // query name of device
               STRING_CLASS name;
               devices[0].getInfo( CL_DEVICE_NAME, &name );
               cout << "Will use this device: " << name << endl;

               // Read kernel source and build a program
               std::string kernel_source = loadProgram("oclHelloVector.ocl");

               Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

               Program program = Program(context, source);
               program.build(devices);

               // Create a kernel
               Kernel kernel(program, "helloVector");

               // Create command queue using the first device
               CommandQueue queue = CommandQueue( context, devices[0], 0 );

               // Create Memory buffers
               Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
               Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
               Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

               // Copy buffers to device
               queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
               queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

               // Create kernel specification (ND range)
               NDRange global(VECT_SIZE);
               NDRange local(1);

               // Set kernel arguments
               kernel.setArg(0, bufA);
               kernel.setArg(1, bufB);
               kernel.setArg(2, bufC);

               // Run kernel
               Event event;
               queue.enqueueNDRangeKernel(   kernel, NullRange, global, local );

               // Copy result buffer from device
               queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

      }
      catch (Error err) {
               cerr << "ERROR: " << err.what()       << "(" << err.err() << ")" << endl;
               error = true;
      }

      // Compare results
      bool test = true;
      for( int i=0; i < VECT_SIZE; i++ ) {
               if (C[i] != VECT_SIZE)
                        test = false;
      }
      if (test)
               cout << "Test PASSED!" << endl;
      else
               cout << "Test FAILED!" << endl;

      delete A;delete B;delete C;
      A = NULL;B = NULL;C = NULL;

      if (error)
               return EXIT_FAILURE;
      else
               return EXIT_SUCCESS;

}

std::string loadProgram(const char *filename) {
      ifstream src(filename);
      std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

      return src_code;
}
```

Compare

Main Program

```
/**
 * This is the Main file for the OpenCL Hello Vector Program.
 *
 * \author: Guillermo Marcus
 * \date: 2011-11-09
 */

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
	const int VECT_SIZE = 100000;
	int *A = new int[VECT_SIZE];
	int *B = new int[VECT_SIZE];
	int *C = new int[VECT_SIZE];
	bool error = false;

	// Fill test data
	for( int i=0; i < VECT_SIZE; i++ ) {
		A[i] = i;
		B[i] = VECT_SIZE - i;
		C[i] = 0;
	}

	try {
		// Check available platforms
		std::vector<Platform> platforms;
		Platform::get( &platforms );

		// Create context
		cl_context_properties cps[3] = {
			CL_CONTEXT_PLATFORM,
			(cl_context_properties)(platforms[0])(),
			0
		};
		Context context( CL_DEVICE_TYPE_GPU, cps );

		// Devices available in this platform
		std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

		// query name of device
		STRING_CLASS name;
		devices[0].getInfo( CL_DEVICE_NAME, &name );
		cout << "Will use this device: " << name << endl;

		// Read kernel source and build a program
		std::string kernel_source = loadProgram("oclHelloVector.ocl");

		Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

		Program program = Program(context, source);
		program.build(devices);

		// Create a kernel
		Kernel kernel(program, "helloVector");

		// Create command queue using the first device
		CommandQueue queue = CommandQueue( context, devices[0], 0 );

		// Create Memory buffers
		Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

		// Copy buffers to device
		queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
		queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

		// Create kernel specification (ND range)
		NDRange global(VECT_SIZE);
		NDRange local(1);

		// Set kernel arguments
		kernel.setArg(0, bufA);
		kernel.setArg(1, bufB);
		kernel.setArg(2, bufC);

		// Run kernel
		Event event;
		queue.enqueueNDRangeKernel(   kernel, NullRange, global, local );

		// Copy result buffer from device
		queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

	}
	catch (Error err) {
		cerr << "ERROR: " << err.what()	<< "(" << err.err() << ")" << endl;
		error = true;
	}

	// Compare results
	bool test = true;
	for( int i=0; i < VECT_SIZE; i++ ) {
		if (C[i] != VECT_SIZE)
			test = false;
	}
	if (test)
		cout << "Test PASSED!" << endl;
	else
		cout << "Test FAILED!" << endl;

	delete A;delete B;delete C;
	A = NULL;B = NULL;C = NULL;

	if (error)
		return EXIT_FAILURE;
	else
		return EXIT_SUCCESS;
}

std::string loadProgram(const char *filename) {
	ifstream src(filename);
	std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

	return src_code;
}
```

Cleanup

Main Program

```
/**
 * This is the Main file for the OpenCL Hello Vector Program.
 *
 * \author: Guillermo Marcus
 * \date: 2011-11-09
 */

#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;

std::string loadProgram(const char *filename);

int main(int argc, char **argv)
{
	const int VECT_SIZE = 100000;
	int *A = new int[VECT_SIZE];
	int *B = new int[VECT_SIZE];
	int *C = new int[VECT_SIZE];
	bool error = false;

	// Fill test data
	for( int i=0; i < VECT_SIZE; i++ ) {
		A[i] = i;
		B[i] = VECT_SIZE - i;
		C[i] = 0;
	}

	try {
		// Check available platforms
		std::vector<Platform> platforms;
		Platform::get( &platforms );

		// Create context
		cl_context_properties cps[3] = {
			CL_CONTEXT_PLATFORM,
			(cl_context_properties)(platforms[0])(),
			0
		};
		Context context( CL_DEVICE_TYPE_GPU, cps );

		// Devices available in this platform
		std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

		// query name of device
		STRING_CLASS name;
		devices[0].getInfo( CL_DEVICE_NAME, &name );
		cout << "Will use this device: " << name << endl;

		// Read kernel source and build a program
		std::string kernel_source = loadProgram("oclHelloVector.ocl");

		Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

		Program program = Program(context, source);
		program.build(devices);

		// Create a kernel
		Kernel kernel(program, "helloVector");

		// Create command queue using the first device
		CommandQueue queue = CommandQueue( context, devices[0], 0 );

		// Create Memory buffers
		Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
		Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

		// Copy buffers to device
		queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
		queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

		// Create kernel specification (ND range)
		NDRange global(VECT_SIZE);
		NDRange local(1);

		// Set kernel arguments
		kernel.setArg(0, bufA);
		kernel.setArg(1, bufB);
		kernel.setArg(2, bufC);

		// Run kernel
		Event event;
		queue.enqueueNDRangeKernel(   kernel, NullRange, global, local );

		// Copy result buffer from device
		queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

	}
	catch (Error err) {
		cerr << "ERROR: " << err.what()	<< "(" << err.err() << ")" << endl;
		error = true;
	}

	// Compare results
	bool test = true;
	for( int i=0; i < VECT_SIZE; i++ ) {
		if (C[i] != VECT_SIZE)
			test = false;
	}
	if (test)
		cout << "Test PASSED!" << endl;
	else
		cout << "Test FAILED!" << endl;

	delete A;delete B;delete C;
	A = NULL;B = NULL;C = NULL;

	if (error)
		return EXIT_FAILURE;
	else
		return EXIT_SUCCESS;

}

std::string loadProgram(const char *filename) {
	ifstream src(filename);
	std::string src_code( istreambuf_iterator<char>(src), (istreambuf_iterator<char>()) );

	return src_code;
}
```

Main - Headers

```
#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;
```

Main - Headers

```
#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;
```

OpenCL C++ Exceptions

Main - Headers

```
#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;
```

OpenCL C++ Exceptions

STL vector

Main - Headers

```
#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;
```

OpenCL C++ Exceptions

STL vector

OpenCL C++ Header

Main - Headers

```
#define __CL_ENABLE_EXCEPTIONS

#include <vector>
#include <CL/cl.hpp>
#include <iostream>
#include <fstream>

using namespace std;
using namespace cl;
```

OpenCL C++ Exceptions

STL vector

OpenCL C++ Header

Namespaces (optional)

Main - Setup

```
const int VECT_SIZE = 100000;
int *A = new int[VECT_SIZE];
int *B = new int[VECT_SIZE];
int *C = new int[VECT_SIZE];
bool error = false;

// Fill test data
for( int i=0; i < VECT_SIZE; i++ ) {
    A[i] = i;
    B[i] = VECT_SIZE - i;
    C[i] = 0;
}
```

Main - Setup

```
const int VECT_SIZE = 100000;
int *A = new int[VECT_SIZE];
int *B = new int[VECT_SIZE];
int *C = new int[VECT_SIZE];
bool error = false;

// Fill test data
for( int i=0; i < VECT_SIZE; i++ ) {
    A[i] = i;
    B[i] = VECT_SIZE - i;
    C[i] = 0;
}
```

Vectors

Main - Setup

```
const int VECT_SIZE = 100000;
int *A = new int[VECT_SIZE];
int *B = new int[VECT_SIZE];
int *C = new int[VECT_SIZE];
bool error = false;

// Fill test data
for( int i=0; i < VECT_SIZE; i++ ) {
    A[i] = i;
    B[i] = VECT_SIZE - i;
    C[i] = 0;
}
```

Vector Size

Vectors

Main - Setup

```
const int VECT_SIZE = 100000;
int *A = new int[VECT_SIZE];
int *B = new int[VECT_SIZE];
int *C = new int[VECT_SIZE];
bool error = false;

// Fill test data
for( int i=0; i < VECT_SIZE; i++ ) {
    A[i] = i;
    B[i] = VECT_SIZE - i;
    C[i] = 0;
}
```

Vector Size

Vectors

Fill vectors with data

Compare & Cleanup

```
// Compare results
bool test = true;
for( int i=0; i < VECT_SIZE; i++ ) {
    if (C[i] != VECT_SIZE)
        test = false;
}
if (test)
    cout << "Test PASSED!" << endl;
else
    cout << "Test FAILED!" << endl;

delete A;
delete B;
delete C;
A = NULL;
B = NULL;
C = NULL;

if (error)
    return EXIT_FAILURE;
else
    return EXIT_SUCCESS;
```

Compare & Cleanup

```
// Compare results
bool test = true;
for( int i=0; i < VECT_SIZE; i++ ) {
    if (C[i] != VECT_SIZE)
        test = false;
}
if (test)
    cout << "Test PASSED!" << endl;
else
    cout << "Test FAILED!" << endl;

delete A;
delete B;
delete C;
A = NULL;
B = NULL;
C = NULL;

if (error)
    return EXIT_FAILURE;
else
    return EXIT_SUCCESS;
```

Compare with expected result

Compare & Cleanup

```
// Compare results
bool test = true;
for( int i=0; i < VECT_SIZE; i++ ) {
    if (C[i] != VECT_SIZE)
        test = false;
}
if (test)
    cout << "Test PASSED!" << endl;
else
    cout << "Test FAILED!" << endl;

delete A;
delete B;
delete C;
A = NULL;
B = NULL;
C = NULL;

if (error)
    return EXIT_FAILURE;
else
    return EXIT_SUCCESS;
```

Compare with expected result

Release vectors

Concepts

- Platform
  - Any Environment that implements OpenCL
- Context
  - Groups Memory mappings to groups of devices and programs
- Device
  - Groups Program Kernels
- Queue
  - Order Tasks (transfers, kernel launches)
- Program
  - Associates code to execute with devices
- Kernel
  - Defines how the program is going to be executed in the device: specification and parameters

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

    }
    catch (Error err) {
            cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
            error = true;
    }
```

Error Handling

Get Platforms

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

GPU Context

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
              CL_CONTEXT_PLATFORM,
              (cl_context_properties)(platforms[0])(),
              0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

GPU Device

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Load & Build Program

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Create Kernel

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Create Queue

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Define GPU Buffers

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Copy Vectors

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Setup and Execute Kernel

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );
}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Copy Result Vector

```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Platform

```
// Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );
```

Intel SDK: CPU only
NVIDIA SDK: GPU only
AMD SDK: CPU and GPU
Apple SDK: CPU and GPU

the ICD will handle the selection of multiple SDKs.

Context

```
// Create context
cl_context_properties cps[3] = {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platforms[0])(),
    0
};
Context context( CL_DEVICE_TYPE_GPU, cps );
```

Context

```
// Create context
cl_context_properties cps[3] = {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platforms[0])(),
    0
};
Context context( CL_DEVICE_TYPE_GPU, cps );
```

"Create a context with GPU devices that belongs to Platform X"

Context

```
// Create context
cl_context_properties cps[3] = {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platforms[0])(),
    0
};
Context context( CL_DEVICE_TYPE_GPU, cps );
```

"Create a context with GPU devices that belongs to Platform X"

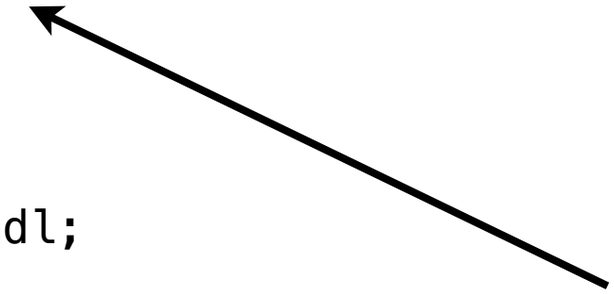
"Create a context for devices of type GPU that also match the following properties: belongs to Platform X"

Devices

```
// Devices available in this platform
std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

// query name of device
STRING_CLASS name;
devices[0].getInfo( CL_DEVICE_NAME, &name );
cout << "Will use this device: " << name << endl;
```

Devices

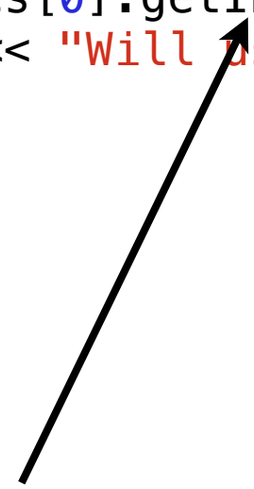
```
// Devices available in this platform
std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

// query name of device
STRING_CLASS name;
devices[0].getInfo( CL_DEVICE_NAME, &name );
cout << "Will use this device: " << name << endl;
```



Ask for devices associated with this context

Devices

```
// Devices available in this platform
std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

// query name of device
STRING_CLASS name;
devices[0].getInfo( CL_DEVICE_NAME, &name );
cout << "Will use this device: " << name << endl;
```



Ask the name of the first device

Devices

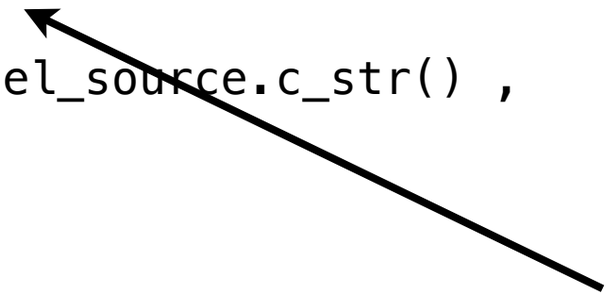
```
// Devices available in this platform
std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

// query name of device
STRING_CLASS name;
devices[0].getInfo( CL_DEVICE_NAME, &name );
cout << "Will use this device: " << name << endl;
```

Load & Build Programs

```
        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");
```

Load & Build Programs

```
        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");
```



Load Program as a std::string

Load & Build Programs

```
        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");
```

Create a Program Source with a string and a size

Load & Build Programs

```
        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");
```

Create a Program, associate source and context

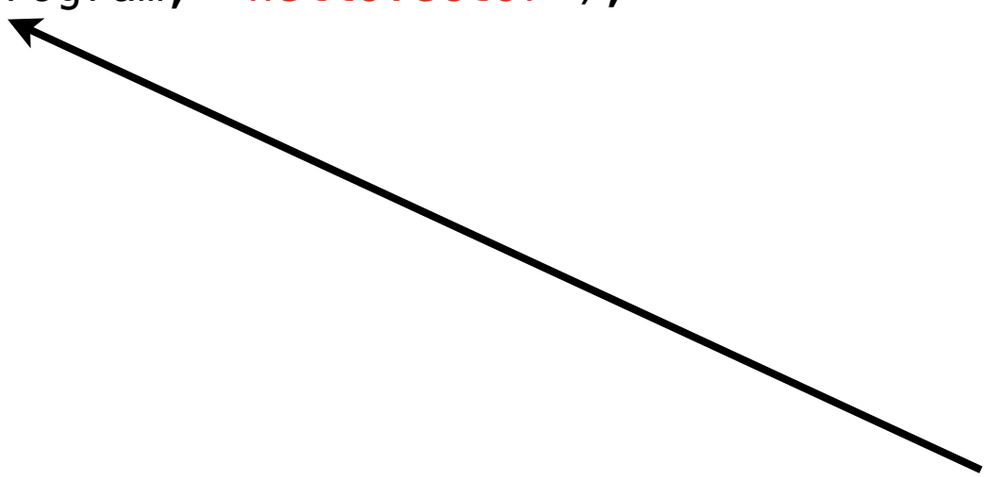
Load & Build Programs

```
        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");
```

Build the program for our devices

Load & Build Programs

```
        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");
```



Create a Kernel object, associate with function "helloVector"

Load & Build Programs

```
        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() ,
kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");
```

Create Command Queue

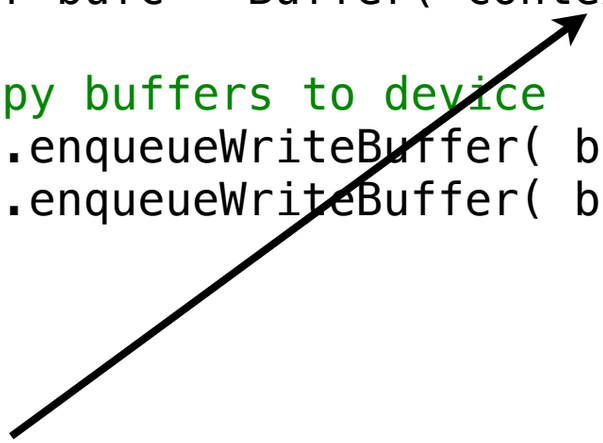
```
// Create command queue using the first device
CommandQueue queue = CommandQueue( context, devices[0], 0 );
```

Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```

Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```



context

Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```

attributes

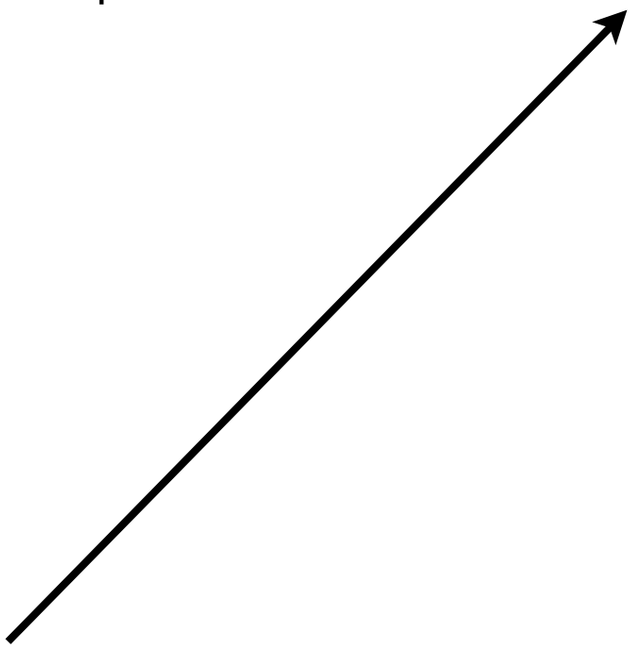
Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```

size, in bytes

Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```



destination

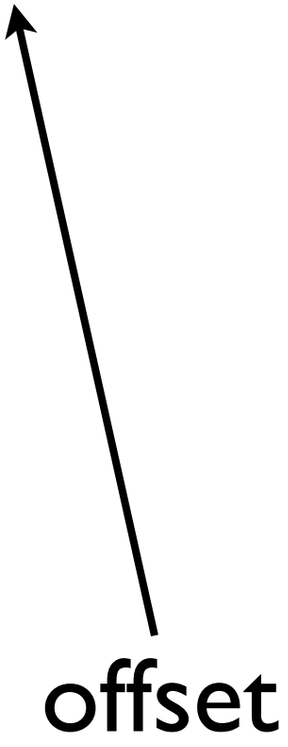
Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```

blocking

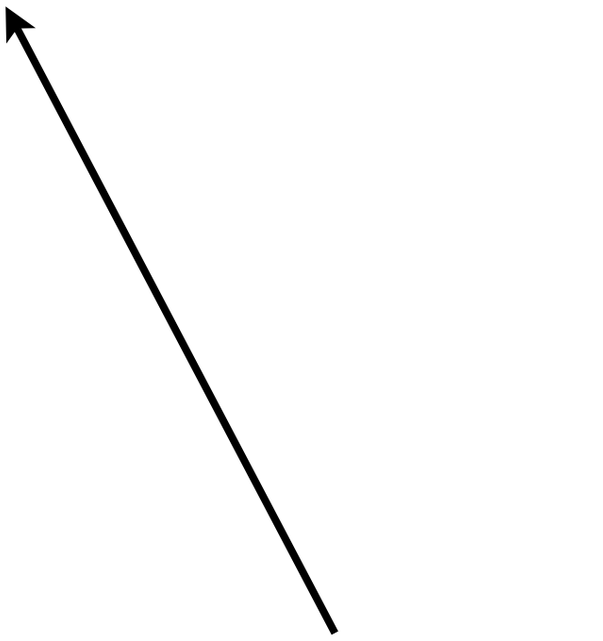
Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```



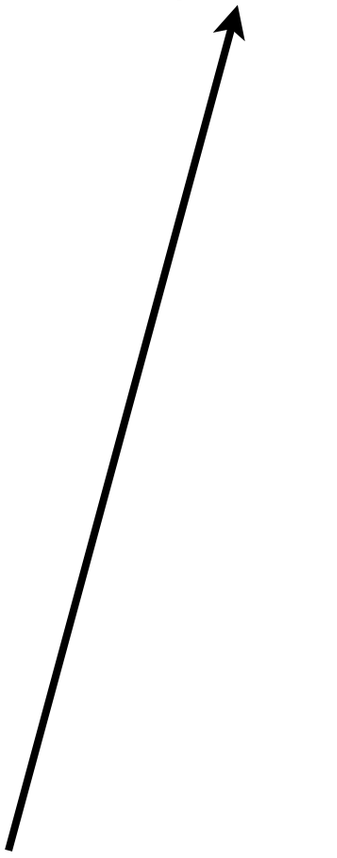
Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```



Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```



source pointer

Buffers

```
// Create Memory buffers
Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

// Copy buffers to device
queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );
```

Kernel Execution

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

Kernel Execution

```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);

// Set kernel arguments
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local );
```

Kernel Execution

```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);

// Set kernel arguments
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local );
```

ranges

Kernel Execution

```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);

// Set kernel arguments
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local );
```

kernel arguments

Kernel Execution

```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);

// Set kernel arguments
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local );
```

Add kernel to execution queue

Kernel Execution

```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);

// Set kernel arguments
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local );
```

offset

Kernel Execution

```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);

// Set kernel arguments
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local );
```

global spec
(workgroups)

Kernel Execution

```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);

// Set kernel arguments
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local );
```

local spec
(work items)

Kernel Execution

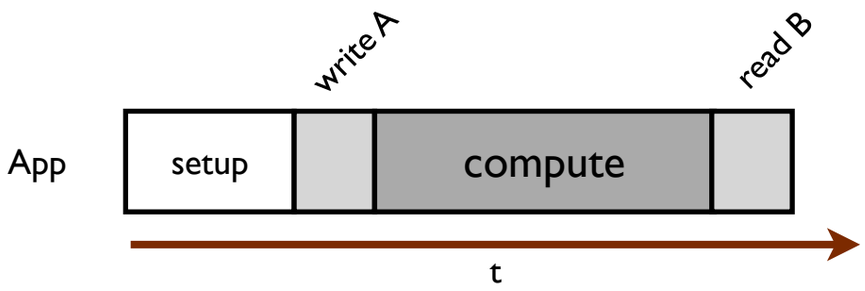
```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);

// Set kernel arguments
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local );
```

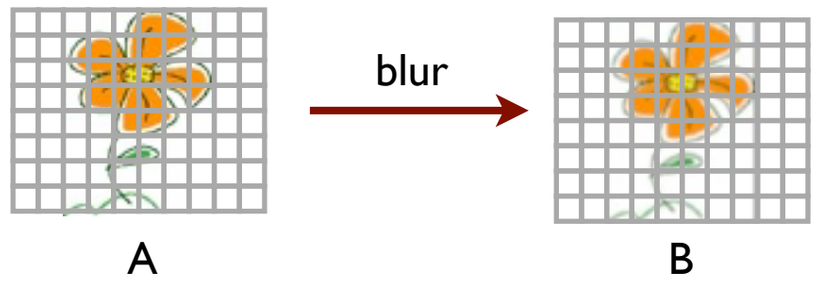
Read Results Back

```
// Copy result buffer from device
queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );
```

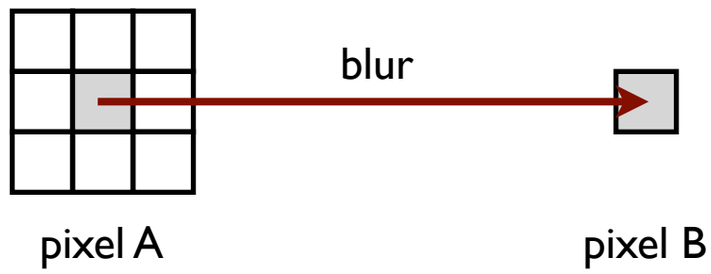
```
try {
        // Check available platforms
        std::vector<Platform> platforms;
        Platform::get( &platforms );

        // Create context
        cl_context_properties cps[3] = {
               CL_CONTEXT_PLATFORM,
               (cl_context_properties)(platforms[0])(),
               0
        };
        Context context( CL_DEVICE_TYPE_GPU, cps );

        // Devices available in this platform
        std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // query name of device
        STRING_CLASS name;
        devices[0].getInfo( CL_DEVICE_NAME, &name );
        cout << "Will use this device: " << name << endl;

        // Read kernel source and build a program
        std::string kernel_source = loadProgram("oclHelloVector.ocl");

        Program::Sources source(1, std::make_pair(kernel_source.c_str() , kernel_source.length()+1 ) );

        Program program = Program(context, source);
        program.build(devices);

        // Create a kernel
        Kernel kernel(program, "helloVector");

        // Create command queue using the first device
        CommandQueue queue = CommandQueue( context, devices[0], 0 );

        // Create Memory buffers
        Buffer bufA = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufB = Buffer( context, CL_MEM_READ_ONLY, VECT_SIZE * sizeof(int) );
        Buffer bufC = Buffer( context, CL_MEM_WRITE_ONLY, VECT_SIZE * sizeof(int) );

        // Copy buffers to device
        queue.enqueueWriteBuffer( bufA, CL_TRUE, 0, VECT_SIZE * sizeof(int), A );
        queue.enqueueWriteBuffer( bufB, CL_TRUE, 0, VECT_SIZE * sizeof(int), B );

        // Create kernel specification (ND range)
        NDRange global(VECT_SIZE);
        NDRange local(1);

        // Set kernel arguments
        kernel.setArg(0, bufA);
        kernel.setArg(1, bufB);
        kernel.setArg(2, bufC);

        // Run kernel
        Event event;
        queue.enqueueNDRangeKernel(      kernel,      NullRange, global, local );

        // Copy result buffer from device
        queue.enqueueReadBuffer( bufC, CL_TRUE, 0, VECT_SIZE * sizeof(int), C );

}
catch (Error err) {
        cerr << "ERROR: " << err.what()  << "(" << err.err() << ")" << endl;
        error = true;
}
```

Example 2: Image blur





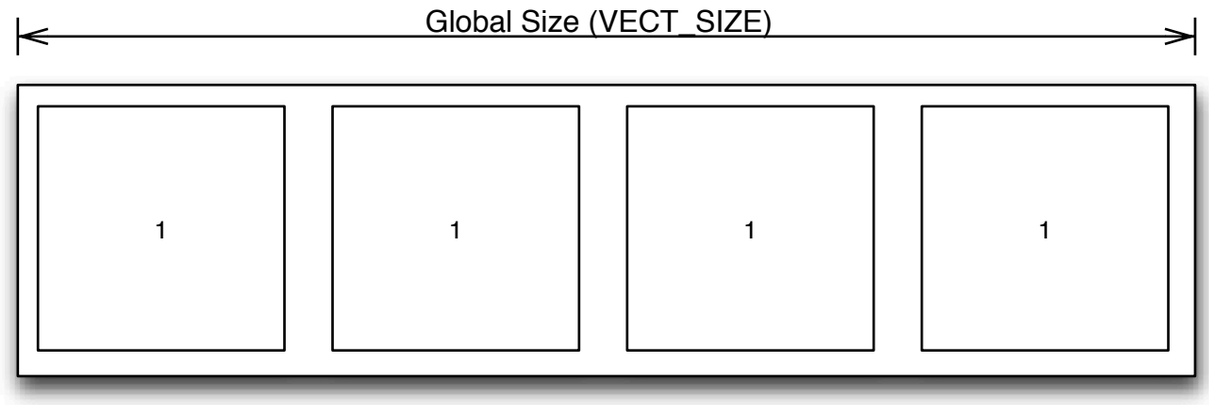
- A simple blur is to make the average of the surrounding pixels
- Every item computes a single element of the destination image B
- Every pixel B requires pixel A and the neighbours of pixel A
- We can use a 2D grid to divide the image in areas, and each area is a work group
- Every group can read the pixels in its area, one per item, and share them with the other items in the group. Note they do not need to share: it will just be more work per item to read more data.



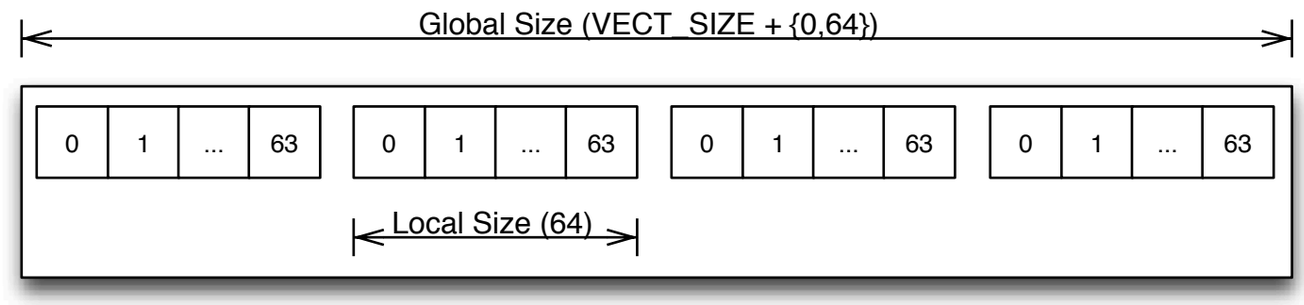

```
int a[w][h], b[w][h];
int n[3][3];

tx = getThreadID(X);
ty = getThreadID(Y);

get_neighbours(tx,ty,n);
b[tx,ty] = average(n);
```

Global and Local Settings

```
// Create kernel specification (ND range)
NDRange global(VECT_SIZE);
NDRange local(1);
```

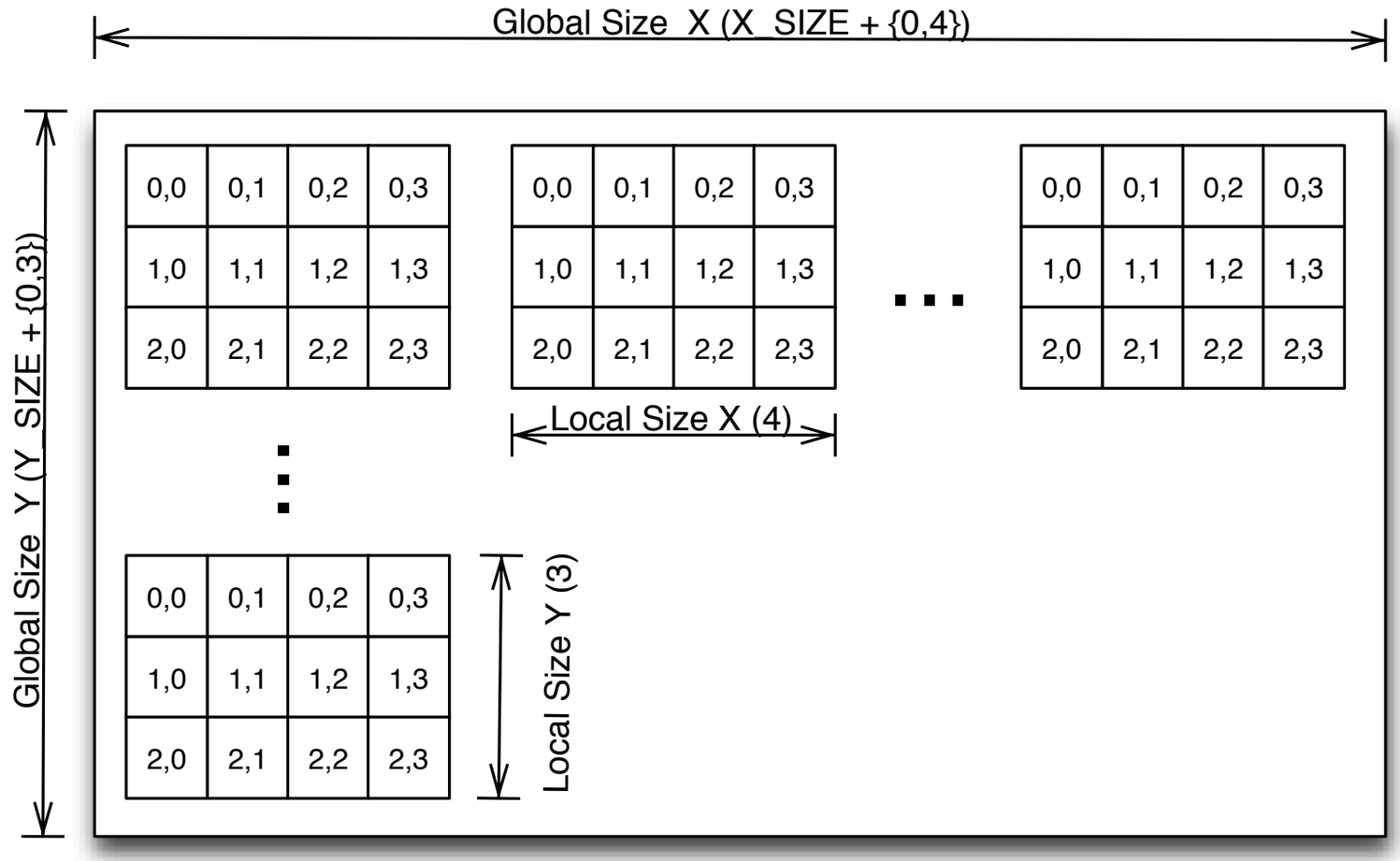



```
// Create kernel specification (ND range)
int groups = VECT_SIZE/64 + ((VECT_SIZE % 64 == 0) ? 0 : 1);
NDRange global(64*groups);
NDRange local(64);
```



Global and Local Settings

```
// Create kernel specification (ND range)
int gX = X_SIZE/4 + ((X_SIZE % 4 == 0) ? 0 : 1);
int gY = Y_SIZE/3 + ((Y_SIZE % 3 == 0) ? 0 : 1);
NDRange global(gX*4, gY*3);
NDRange local(4,3);
```



Kernel Code

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

Kernel Code

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

Address Space Qualifiers

```
__global
__local
__constant
__private
```

Kernel Code

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

Address Space Qualifiers

- `__global`
- `__local`
- `__constant`
- `__private`

Function Qualifier

- `__kernel`

Kernel Code

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

Address Space Qualifiers

```
__global
__local
__constant
__private
```

Function Qualifier

```
__kernel
```

Basic Built-in Functions

```
uint get_work_dim();

size_t get_global_id(uint D);
size_t get_local_id(uint D);

size_t get_global_size(uint D);
size_t get_local_size(uint D);

size_t get_num_groups(uint D);
size_t get_group_id(uint D);
```

Kernel Code

```
__kernel void helloVector(__global const int *A, __global const int *B, __global int *C) {
   int i = get_global_id(0);

   C[i] = A[i] + B[i];
   return;
}
```

Address Space Qualifiers

```
__global
__local
__constant
__private
```

Function Qualifier

```
__kernel
```

Basic Built-in Functions

```
uint get_work_dim();

size_t get_global_id(uint D);
size_t get_local_id(uint D);

size_t get_global_size(uint D);
size_t get_local_size(uint D);

size_t get_num_groups(uint D);
size_t get_group_id(uint D);
```

Sync and Fences Functions

```
void barrier(cl_mem_fence_flags flags);

void mem_fence(cl_mem_fence_flags flags);
void read_mem_fence(cl_mem_fence_flags flags);
void write_mem_fence(cl_mem_fence_flags flags);
```

Address Space Qualifiers

Address Space Qualifiers

`__global`

Refers to data in the global memory.
Shared with all threads in all work groups.
Memory is not coherent. Needs explicit synchronization.

Address Space Qualifiers

`__global`

Refers to data in the global memory.
Shared with all threads in all work groups.
Memory is not coherent. Needs explicit synchronization.

`__local`

Refers to data in the local memory (local to each work group).
Shared with all threads inside the work groups.
Memory is not coherent. Needs explicit synchronization.

Address Space Qualifiers

`__global`

Refers to data in the global memory.
Shared with all threads in all work groups.
Memory is not coherent. Needs explicit synchronization.

`__local`

Refers to data in the local memory (local to each work group).
Shared with all threads inside the work groups.
Memory is not coherent. Needs explicit synchronization.

`__constant`

Refers to data in the constant memory. Read Only.
Shared with all threads in all work groups.

Address Space Qualifiers

`__global`

Refers to data in the global memory.
Shared with all threads in all work groups.
Memory is not coherent. Needs explicit synchronization.

`__local`

Refers to data in the local memory (local to each work group).
Shared with all threads inside the work groups.
Memory is not coherent. Needs explicit synchronization.

`__constant`

Refers to data in the constant memory. Read Only.
Shared with all threads in all work groups.

`__private`

Refers to data in the private memory.
Data is not shared (private to the work item).
Memory is coherent. No explicit synchronization needed.

Function Qualifier

`__kernel`

Defines a Kernel function, that can be invoked from the host.
Cannot return a value.
Has limitations in the parameters it can receive: pointers to main memory, constants, single data values.

Any other function is a local function and must be called from inside a Kernel function.

Comply with most C99 standard.

Basic Built-in Functions

Basic Built-in Functions

```
uint get_work_dim();
```

Returns the number of dimensions in use.

Basic Built-in Functions

```
uint get_work_dim();
```

Returns the number of dimensions in use.

```
size_t get_global_id(uint D);
```

Returns the global ID of the work item for dimension D.

This number is unique among all items in the Kernel.

Basic Built-in Functions

`uint get_work_dim();`

Returns the number of dimensions in use.

`size_t get_global_id(uint D);`

Returns the global ID of the work item for dimension D.

This number is unique among all items in the Kernel.

`size_t get_local_id(uint D);`

Returns the local ID of the work item inside the work group for dimension D.

This number is unique only inside the Workgroup.

Basic Built-in Functions

`uint get_work_dim();`

Returns the number of dimensions in use.

`size_t get_global_id(uint D);`

Returns the global ID of the work item for dimension D.

This number is unique among all items in the Kernel.

`size_t get_local_id(uint D);`

Returns the local ID of the work item inside the work group for dimension D.

This number is unique only inside the Workgroup.

`size_t get_global_size(uint D);`

Returns the number of work items in the dimension D.

Basic Built-in Functions

`uint get_work_dim();`

Returns the number of dimensions in use.

`size_t get_global_id(uint D);`

Returns the global ID of the work item for dimension D.

This number is unique among all items in the Kernel.

`size_t get_local_id(uint D);`

Returns the local ID of the work item inside the work group for dimension D.

This number is unique only inside the Workgroup.

`size_t get_global_size(uint D);`

Returns the number of work items in the dimension D.

`size_t get_local_size(uint D);`

Returns the number of work items in the workgroup in the dimension D.

Basic Built-in Functions

`uint get_work_dim();`

Returns the number of dimensions in use.

`size_t get_global_id(uint D);`

Returns the global ID of the work item for dimension D.

This number is unique among all items in the Kernel.

`size_t get_local_id(uint D);`

Returns the local ID of the work item inside the work group for dimension D.

This number is unique only inside the Workgroup.

`size_t get_global_size(uint D);`

Returns the number of work items in the dimension D.

`size_t get_local_size(uint D);`

Returns the number of work items in the workgroup in the dimension D.

`size_t get_num_groups(uint D);`

Returns the total number of Workgroups in the dimension D.

Basic Built-in Functions

`uint get_work_dim();`

Returns the number of dimensions in use.

`size_t get_global_id(uint D);`

Returns the global ID of the work item for dimension D.

This number is unique among all items in the Kernel.

`size_t get_local_id(uint D);`

Returns the local ID of the work item inside the work group for dimension D.

This number is unique only inside the Workgroup.

`size_t get_global_size(uint D);`

Returns the number of work items in the dimension D.

`size_t get_local_size(uint D);`

Returns the number of work items in the workgroup in the dimension D.

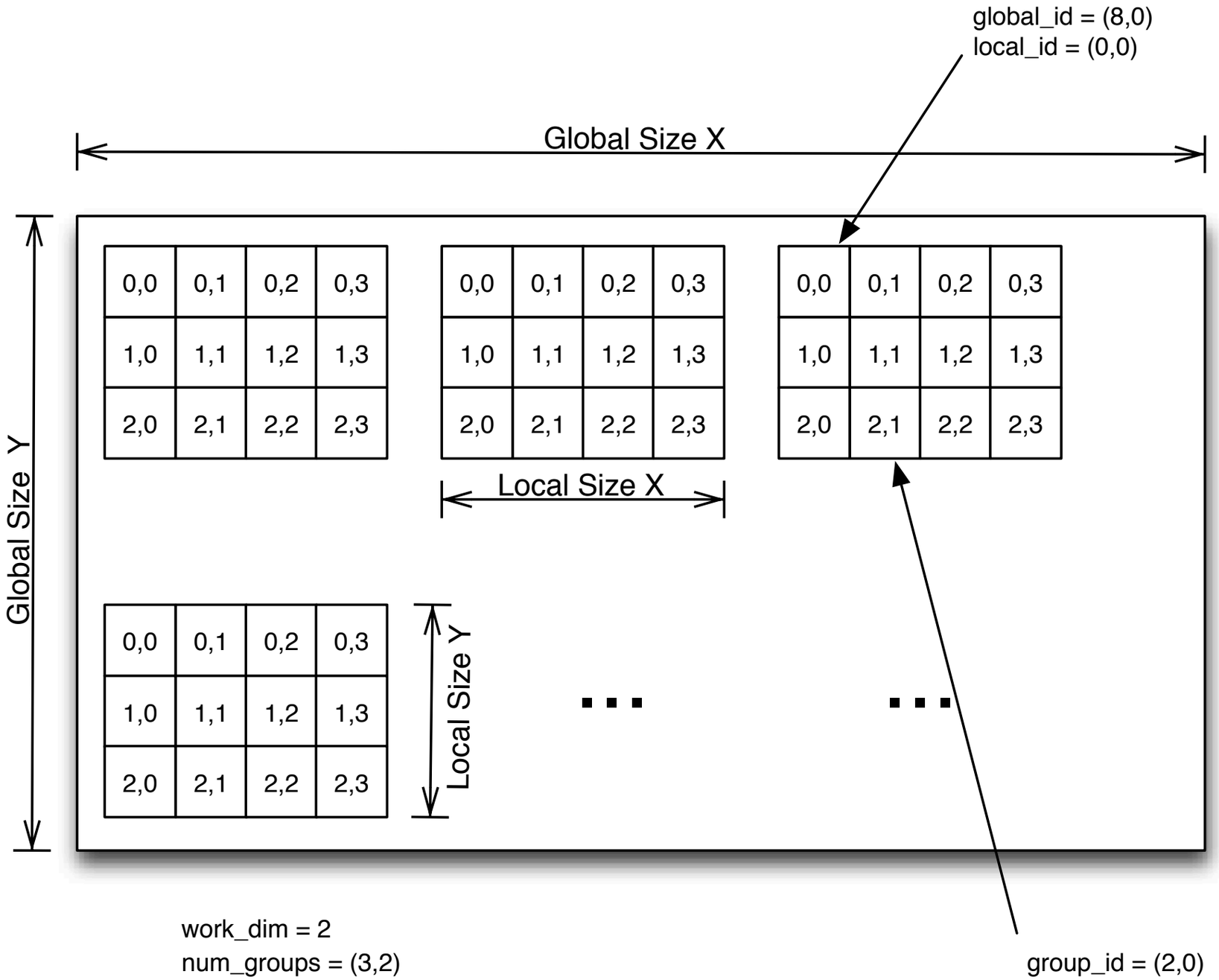
`size_t get_num_groups(uint D);`

Returns the total number of Workgroups in the dimension D.

`size_t get_group_id(uint D);`

Returns the ID of the Workgroup in the dimension D.

Basic Built-in Functions



Sync Barriers and Fences

Sync Barriers and Fences

```
void barrier(cl_mem_fence_flags flags);
```

All work items must execute this function before continuing.

Sync Barriers and Fences

```
void barrier(cl_mem_fence_flags flags);
```

All work items must execute this function before continuing.

```
void mem_fence(cl_mem_fence_flags flags);
```

Orders memory accesses. All memory accesses (load and stores) will be **committed** to memory before memory accesses after the fence.

Sync Barriers and Fences

```
void barrier(cl_mem_fence_flags flags);
```

All work items must execute this function before continuing.

```
void mem_fence(cl_mem_fence_flags flags);
```

Orders memory accesses. All memory accesses (load and stores) will be **committed** to memory before memory accesses after the fence.

```
void read_mem_fence(cl_mem_fence_flags flags);
```

Orders only memory loads.

Sync Barriers and Fences

```
void barrier(cl_mem_fence_flags flags);
```

All work items must execute this function before continuing.

```
void mem_fence(cl_mem_fence_flags flags);
```

Orders memory accesses. All memory accesses (load and stores) will be **committed** to memory before memory accesses after the fence.

```
void read_mem_fence(cl_mem_fence_flags flags);
```

Orders only memory loads.

```
void write_mem_fence(cl_mem_fence_flags flags);
```

Orders only memory stores.

Sync Barriers and Fences

```
void barrier(cl_mem_fence_flags flags);
```

All work items must execute this function before continuing.

```
void mem_fence(cl_mem_fence_flags flags);
```

Orders memory accesses. All memory accesses (load and stores) will be **committed** to memory before memory accesses after the fence.

```
void read_mem_fence(cl_mem_fence_flags flags);
```

Orders only memory loads.

```
void write_mem_fence(cl_mem_fence_flags flags);
```

Orders only memory stores.

```
CLK_LOCAL_MEM_FENCE
```

Apply the barrier/fence to local memory accesses

Sync Barriers and Fences

```
void barrier(cl_mem_fence_flags flags);
```

All work items must execute this function before continuing.

```
void mem_fence(cl_mem_fence_flags flags);
```

Orders memory accesses. All memory accesses (load and stores) will be **committed** to memory before memory accesses after the fence.

```
void read_mem_fence(cl_mem_fence_flags flags);
```

Orders only memory loads.

```
void write_mem_fence(cl_mem_fence_flags flags);
```

Orders only memory stores.

`CLK_LOCAL_MEM_FENCE`

Apply the barrier/fence to local memory accesses

`CLK_GLOBAL_MEM_FENCE`

Apply the barrier/fence to global memory accesses

Sync Barriers and Fences

Sync Barriers and Fences

Barrier

```
__local int d[256];
parallel_sort( d );
barrier( CLK_LOCAL_MEM_FENCE );
parallel_select( max, d );
```

Sync Barriers and Fences

Barrier

```
__local int d[256];
parallel_sort( d );
barrier( CLK_LOCAL_MEM_FENCE );
parallel_select( max, d );
```

Mem Fence (RW)

```
while(1) {
    prefetch_new_input( d );
    mem_fence( CLK_GLOBAL_MEM_FENCE );
    x = process_data( d );
    save_batch( x );
    if (exit_code) break;
}
```

Sync Barriers and Fences

Barrier

```
__local int d[256];
parallel_sort( d );
barrier( CLK_LOCAL_MEM_FENCE );
parallel_select( max, d );
```

Mem Fence (RW)

```
while(1) {
    prefetch_new_input( d );
    mem_fence( CLK_GLOBAL_MEM_FENCE );
    x = process_data( d );
    save_batch( x );
    if (exit_code) break;
}
```

Read Fence

```
__local int d[256];
prefetch( d );
mem_read_fence( CLK_GLOBAL_MEM_FENCE );
```

Sync Barriers and Fences

Barrier

```
__local int d[256];
parallel_sort( d );
barrier( CLK_LOCAL_MEM_FENCE );
parallel_select( max, d );
```

Mem Fence (RW)

```
while(1) {
    prefetch_new_input( d );
    mem_fence( CLK_GLOBAL_MEM_FENCE );
    x = process_data( d );
    save_batch( x );
    if (exit_code) break;
}
```

Read Fence

```
__local int d[256];
prefetch( d );
mem_read_fence( CLK_GLOBAL_MEM_FENCE );
```

Write Fence

```
__local int d[256];
__global int* gD;
compute_SPH_step1( gD );
compute_EOS(d);
gD[ get_global_id(0) ] = d[ get_local_id(0) ];
mem_write_fence( CLK_GLOBAL_MEM_FENCE );
compute_SPH_step2( gD );
```

OpenCL Events

- Events are objects to track the execution of a command
- Events provide information about the execution of commands (i.e. profiling)
- Events provide synchronization and ordering to sequences of commands (beyond a single queue)

Event Functions

- cl::Event::getInfo(name, param)
- cl::Event::getProfilingInfo(name, param)
- cl::Event::wait()
- cl::Event::waitForEvents(list_of_events)

cl::Event::getInfo

```
cl_int cl::Event::getInfo(cl_event_info name, T* param)
```

cl_event_info	Return Type
CL_EVENT_CONTEXT	Cl::Context
CL_EVENT_COMMAND_QUEUE	cl::CommandQueue

Table 9: Difference in return type for table 5.15 and cl::Event::getInfo

cl_event_info	**Return Type**	**Info. returned in *param_value***
CL_EVENT_COMMAND_ QUEUE	cl_command_ queue	Return the command-queue associated with *event*.
CL_EVENT_COMMAND_ TYPE	cl_command_type	Return the command associated with event. Can be one of the following values: CL_COMMAND_NDRANGE_KERNEL CL_COMMAND_TASK CL_COMMAND_NATIVE_KERNEL CL_COMMAND_READ_BUFFER CL_COMMAND_WRITE_BUFFER CL_COMMAND_COPY_BUFFER CL_COMMAND_READ_IMAGE CL_COMMAND_WRITE_IMAGE CL_COMMAND_COPY_IMAGE CL_COMMAND_COPY_BUFFER_TO_IMAGE CL_COMMAND_COPY_IMAGE_TO_BUFFER CL_COMMAND_MAP_BUFFER CL_COMMAND_MAP_IMAGE CL_COMMAND_UNMAP_MEM_OBJECT CL_COMMAND_MARKER CL_COMMAND_ACQUIRE_GL_OBJECTS CL_COMMAND_RELEASE_GL_OBJECTS
CL_EVENT_COMMAND_ EXECUTION_STATUS	cl_int	Return the execution status of the command identified by *event*. Valid values are: CL_QUEUED (command has been enqueued in the command-queue), CL_SUBMITTED (enqueued command has been submitted by the host to the device associated with the command-queue), CL_RUNNING (device is currently executing this command), CL_COMPLETE (the command has completed), or Error code given by a negative integer value. (command was abnormally terminated – this may be caused by a bad memory access etc.).
CL_EVENT_REFERENCE_ COUNT[10]	cl_uint	Return the *event* reference count.

Table 5.15 *clGetEventInfo parameter queries.*

Example

```
// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event );
event.wait();
```

Example

```
// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event );
event.wait();
```

Event definition

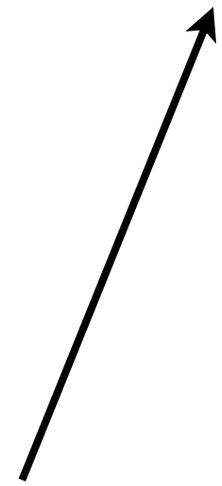
Example

```
// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event );
event.wait();
```

list of events to wait for

Example

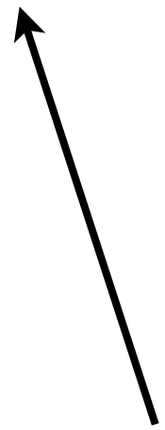
```
// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event );
event.wait();
```



event associated
with this command

Example

```
// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event );
event.wait();
```



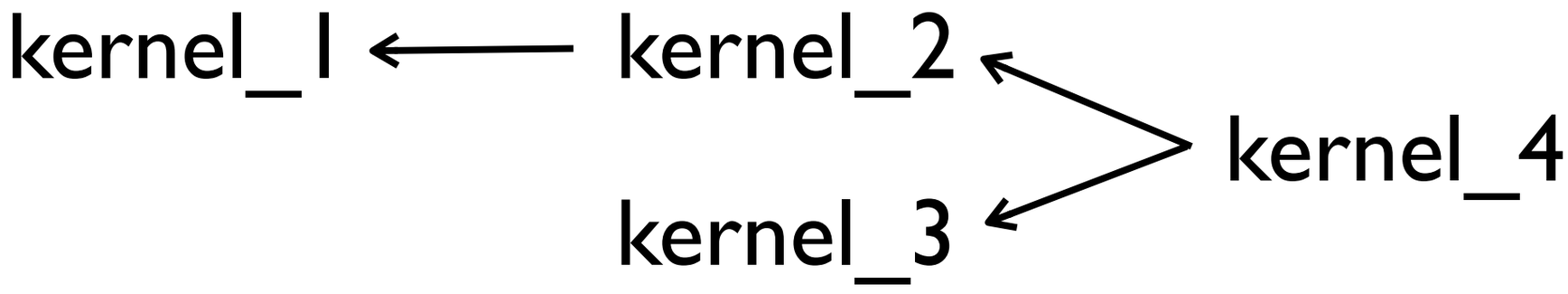
wait for the event

Example

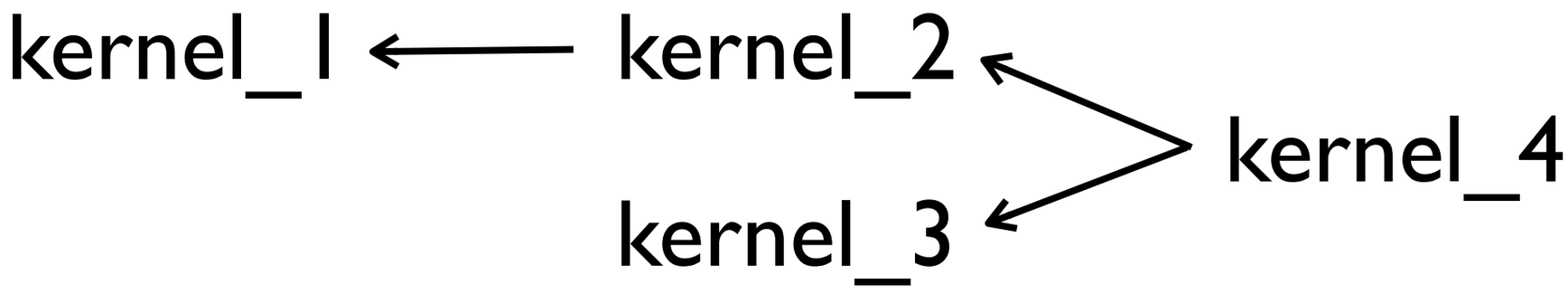
```
// Run kernel
Event event;
queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event );
event.wait();
```

Example II

Example II



Example II




```
// Run kernel
      Event event1, event2, event3, event4;

      std::vector<Event> list1, list2;

      list1.push_back( event1 );
      list2.push_back( event2 );
      list2.push_back( event3 );

      queue.enqueueNDRangeKernel(   kernel_1, NullRange, global, local, NULL, &list1[0] );
      queue.enqueueNDRangeKernel(   kernel_2, NullRange, global, local, &list1, &list2[0] );
      queue.enqueueNDRangeKernel(   kernel_3, NullRange, global, local, NULL, &list2[1] );
      queue.enqueueNDRangeKernel(   kernel_4, NullRange, global, local, &list2, &event4 );
      event4.wait();
```

Example III - Barrier and Finish

Example III - Barrier and Finish

```
// Run kernel
      Event event1, event2;

      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event1 );
      event1.wait();
      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event2 );
      event2.wait();
```

Example III - Barrier and Finish

```
// Run kernel
      Event event1, event2;

      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event1 );
      event1.wait();
      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event2 );
      event2.wait();
```

```
// Run kernel
      Event event1, event2;
      std::vector<Event> list1;

      list1.push_back( event1 );

      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &list1[0] );
      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, &list1, &event2 );
      event2.wait();
```

Example III - Barrier and Finish

```
// Run kernel
      Event event1, event2;

      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event1 );
      event1.wait();
      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event2 );
      event2.wait();
```

```
// Run kernel
      Event event1, event2;
      std::vector<Event> list1;

      list1.push_back( event1 );

      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &list1[0] );
      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, &list1, &event2 );
      event2.wait();
```

```
// Run kernel
       queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, NULL );
       queue.enqueueBarrier();
       queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, NULL );
       queue.finish();
```

Example IV - Multiple queues

```
// Create command queue using the first device
CommandQueue queue = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );
CommandQueue queue_rw = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );

// Create Memory buffers
...

// Copy buffers to device
Event ev1, ev2;
std::vector<Event> ev_io;
ev_io.push_back( ev1 );
ev_io.push_back( ev2 );

queue_rw.enqueueWriteBuffer( bufA, CL_FALSE, 0, VECT_SIZE * sizeof(int), A, NULL, &ev_io[0] );
queue_rw.enqueueWriteBuffer( bufB, CL_FALSE, 0, VECT_SIZE * sizeof(int), B ,NULL, &ev_io[1] );

// Create kernel specification (ND range)
...

// Run kernel
Event event;

queue.enqueueNDRangeKernel(   kernel,  NullRange, global, local, &ev_io, &event );
queue.finish();
```

Example IV - Multiple queues

```
// Create command queue using the first device
CommandQueue queue = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );
CommandQueue queue_rw = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );

// Create Memory buffers
...

// Copy buffers to device
Event ev1, ev2;
std::vector<Event> ev_io;
ev_io.push_back( ev1 );
ev_io.push_back( ev2 );

queue_rw.enqueueWriteBuffer( bufA, CL_FALSE, 0, VECT_SIZE * sizeof(int), A, NULL, &ev_io[0] );
queue_rw.enqueueWriteBuffer( bufB, CL_FALSE, 0, VECT_SIZE * sizeof(int), B ,NULL, &ev_io[1] );

// Create kernel specification (ND range)
...

// Run kernel
Event event;

queue.enqueueNDRangeKernel(   kernel,  NullRange, global, local, &ev_io, &event );
queue.finish();
```

second queue, used for IO only

Example IV - Multiple queues

```
// Create command queue using the first device
CommandQueue queue = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );
CommandQueue queue_rw = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );

// Create Memory buffers
...

// Copy buffers to device
Event ev1, ev2;
std::vector<Event> ev_io;
ev_io.push_back( ev1 );
ev_io.push_back( ev2 );

queue_rw.enqueueWriteBuffer( bufA, CL_FALSE, 0, VECT_SIZE * sizeof(int), A, NULL, &ev_io[0] );
queue_rw.enqueueWriteBuffer( bufB, CL_FALSE, 0, VECT_SIZE * sizeof(int), B ,NULL, &ev_io[1] );

// Create kernel specification (ND range)
...

// Run kernel
Event event;

queue.enqueueNDRangeKernel(   kernel,  NullRange, global, local, &ev_io, &event );
queue.finish();
```

event list

Example IV - Multiple queues

```
// Create command queue using the first device
CommandQueue queue = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );
CommandQueue queue_rw = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );

// Create Memory buffers
...

// Copy buffers to device
Event ev1, ev2;
std::vector<Event> ev_io;
ev_io.push_back( ev1 );
ev_io.push_back( ev2 );

queue_rw.enqueueWriteBuffer( bufA, CL_FALSE, 0, VECT_SIZE * sizeof(int), A, NULL, &ev_io[0] );
queue_rw.enqueueWriteBuffer( bufB, CL_FALSE, 0, VECT_SIZE * sizeof(int), B ,NULL, &ev_io[1] );

// Create kernel specification (ND range)
...

// Run kernel
Event event;

queue.enqueueNDRangeKernel(   kernel,  NullRange, global, local, &ev_io, &event );
queue.finish();
```

non-blocking!

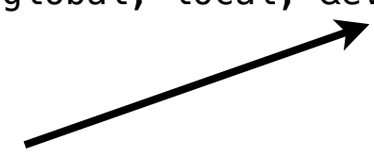
Example IV - Multiple queues

```
// Create command queue using the first device
CommandQueue queue = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );
CommandQueue queue_rw = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );

// Create Memory buffers
...

// Copy buffers to device
Event ev1, ev2;
std::vector<Event> ev_io;
ev_io.push_back( ev1 );
ev_io.push_back( ev2 );

queue_rw.enqueueWriteBuffer( bufA, CL_FALSE, 0, VECT_SIZE * sizeof(int), A, NULL, &ev_io[0] );
queue_rw.enqueueWriteBuffer( bufB, CL_FALSE, 0, VECT_SIZE * sizeof(int), B ,NULL, &ev_io[1] );

// Create kernel specification (ND range)
...

// Run kernel
Event event;

queue.enqueueNDRangeKernel(   kernel,  NullRange, global, local, &ev_io, &event );
queue.finish();
```

associate events

Example IV - Multiple queues

```
// Create command queue using the first device
CommandQueue queue = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );
CommandQueue queue_rw = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );

// Create Memory buffers
...

// Copy buffers to device
Event ev1, ev2;
std::vector<Event> ev_io;
ev_io.push_back( ev1 );
ev_io.push_back( ev2 );

queue_rw.enqueueWriteBuffer( bufA, CL_FALSE, 0, VECT_SIZE * sizeof(int), A, NULL, &ev_io[0] );
queue_rw.enqueueWriteBuffer( bufB, CL_FALSE, 0, VECT_SIZE * sizeof(int), B ,NULL, &ev_io[1] );

// Create kernel specification (ND range)
...

// Run kernel
Event event;

queue.enqueueNDRangeKernel(   kernel,  NullRange, global, local, &ev_io, &event );
queue.finish();
```



wait for IO to be completed

Example IV - Multiple queues

```
// Create command queue using the first device
CommandQueue queue = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );
CommandQueue queue_rw = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );

// Create Memory buffers
...

// Copy buffers to device
Event ev1, ev2;
std::vector<Event> ev_io;
ev_io.push_back( ev1 );
ev_io.push_back( ev2 );

queue_rw.enqueueWriteBuffer( bufA, CL_FALSE, 0, VECT_SIZE * sizeof(int), A, NULL, &ev_io[0] );
queue_rw.enqueueWriteBuffer( bufB, CL_FALSE, 0, VECT_SIZE * sizeof(int), B ,NULL, &ev_io[1] );

// Create kernel specification (ND range)
...

// Run kernel
Event event;

queue.enqueueNDRangeKernel(   kernel,  NullRange, global, local, &ev_io, &event );
queue.finish();
```

cl::Event::getProfilingInfo

```
cl_int cl::Event::getProfilingInfo(cl_profiling_info name, T* param)
```

cl_profiling_info	Return Type	Info. returned in *param_value*
CL_PROFILING_COMMAND_QUEUED	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* is enqueued in a command-queue by the host.
CL_PROFILING_COMMAND_SUBMIT	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* that has been enqueued is submitted by the host to the device associated with the command-queue.
CL_PROFILING_COMMAND_START	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* starts execution on the device.
CL_PROFILING_COMMAND_END	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by *event* has finished execution on the device.

Table 5.16 *clGetEventProfilingInfo parameter queries.*

Example

```
    // Create command queue using the first device
    CommandQueue queue = CommandQueue( context, devices[0], CL_QUEUE_PROFILING_ENABLE );



// Run kernel
      Event event;

      queue.enqueueNDRangeKernel(   kernel,   NullRange, global, local, NULL, &event );
      event.wait();

      cl_ulong queued, submit, start, end;

      event.getProfilingInfo( CL_PROFILING_COMMAND_QUEUED, &queued );
      event.getProfilingInfo( CL_PROFILING_COMMAND_SUBMIT, &submit );
      event.getProfilingInfo( CL_PROFILING_COMMAND_START, &start );
      event.getProfilingInfo( CL_PROFILING_COMMAND_END, &end );

      cout << "QUEUED: " << queued << endl
      << "SUBMIT: " << submit << endl
      << "START: " << start << endl
      << "END: " << end << endl
      << "runtime: " << (end - start) *1.e-9 << " secs" << endl;
```

N-Body in GPUs

- Generic form
  - F_{ij} is the gravity interaction of particles i and j
  - Many particles (millions), may not fit in GPU memory
  - Two cases:
    - all particles actives: i-particle set = j-particle set
    - small set active: i-particle is a subset of j-particle, #i << #j

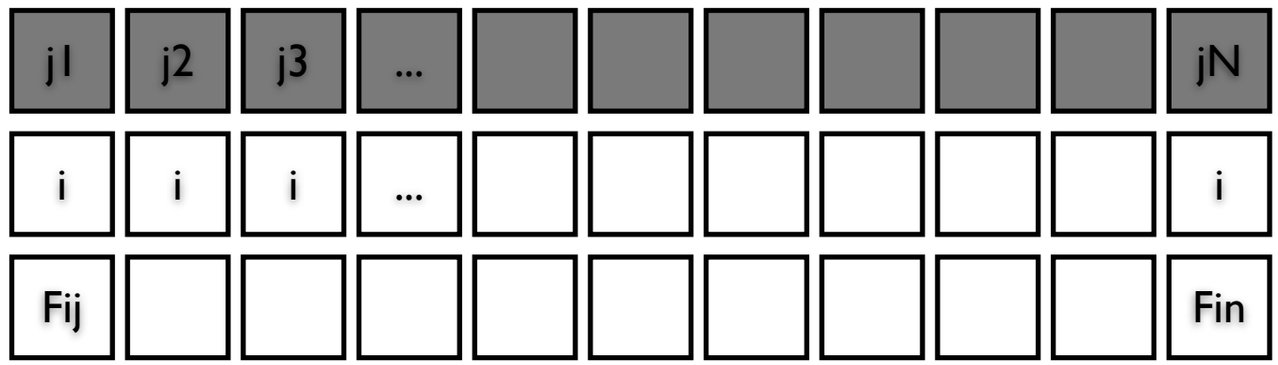
$$x_i = \sum_j f_{ij}$$

GPU implementation options

- One interaction (F_{ij}) per thread. Reduce results (summation) afterwards.
- One i-particle per thread. Fetch j-particles from main memory each time.
- One i-particle per thread. Fetch batch of j-particles and process all before moving to next batch.

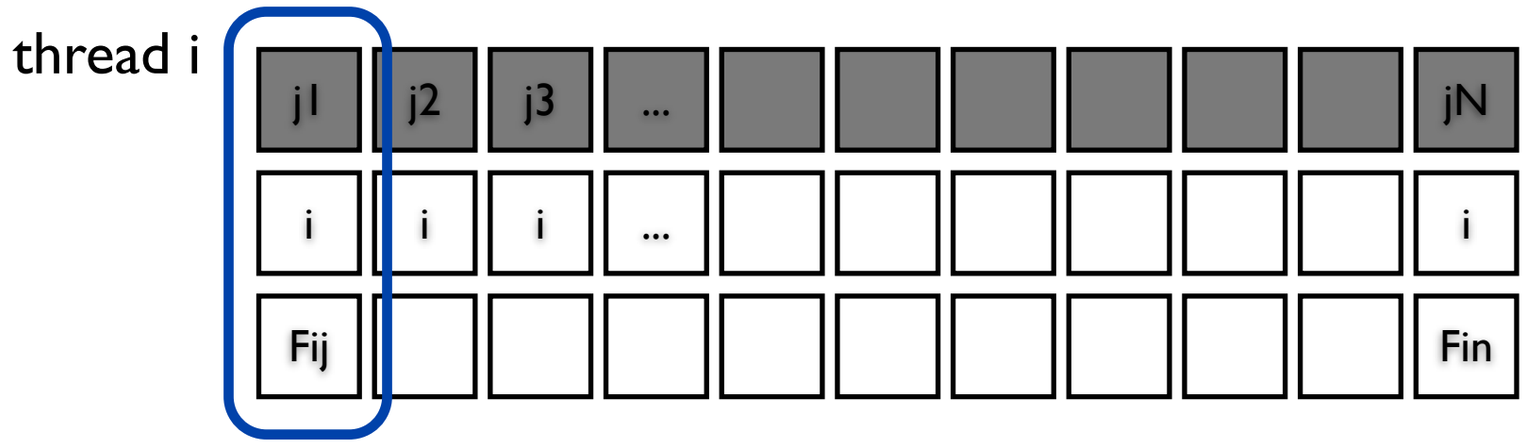
One interaction per thread

- Easy to implement
  - Reuse reduction algorithm
- Wasteful:
  - Need to store interactions before reduction
  - particle data is used once
  - need to repeat for each i-particle.



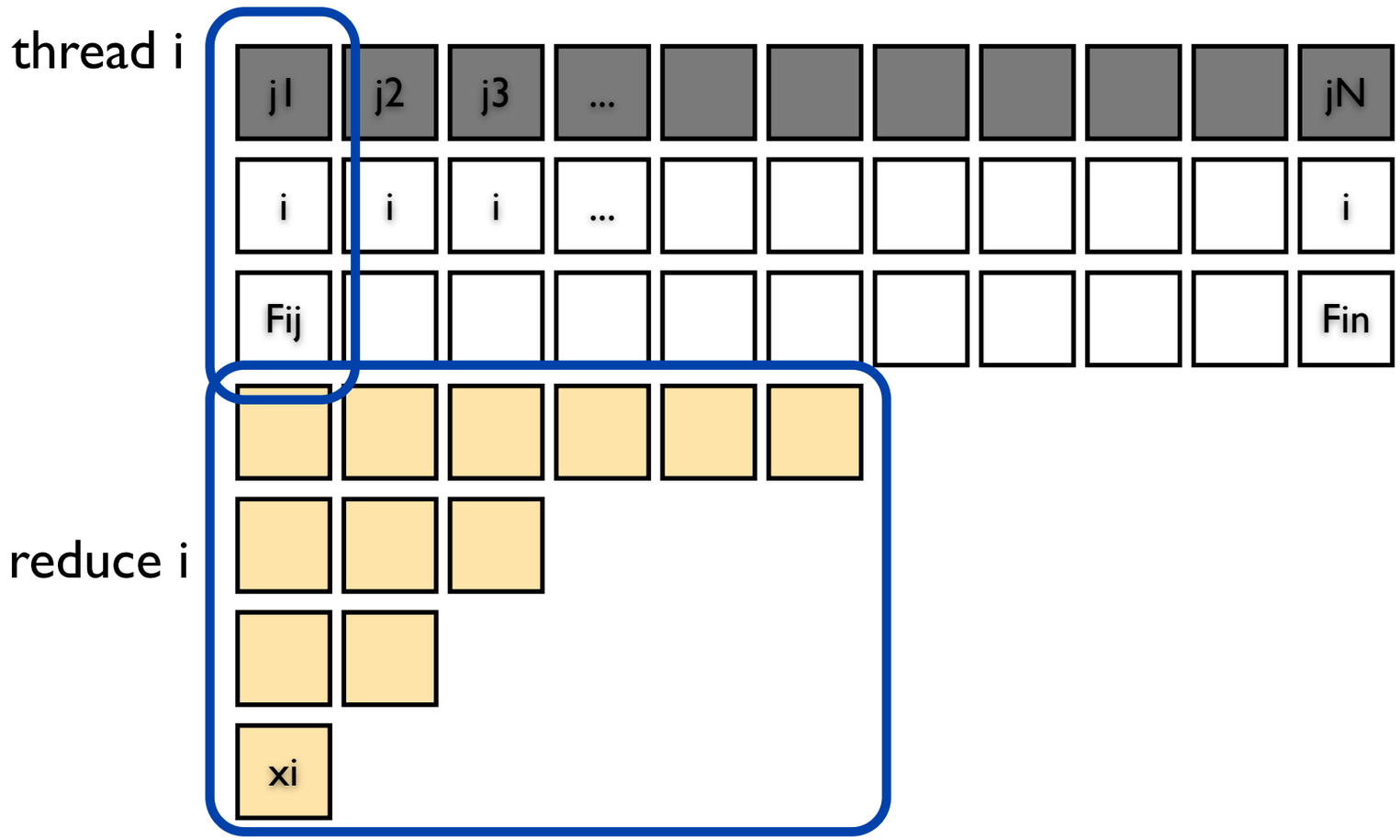
One interaction per thread

- Easy to implement
  - Reuse reduction algorithm
- Wasteful:
  - Need to store interactions before reduction
  - particle data is used once
  - need to repeat for each i-particle.



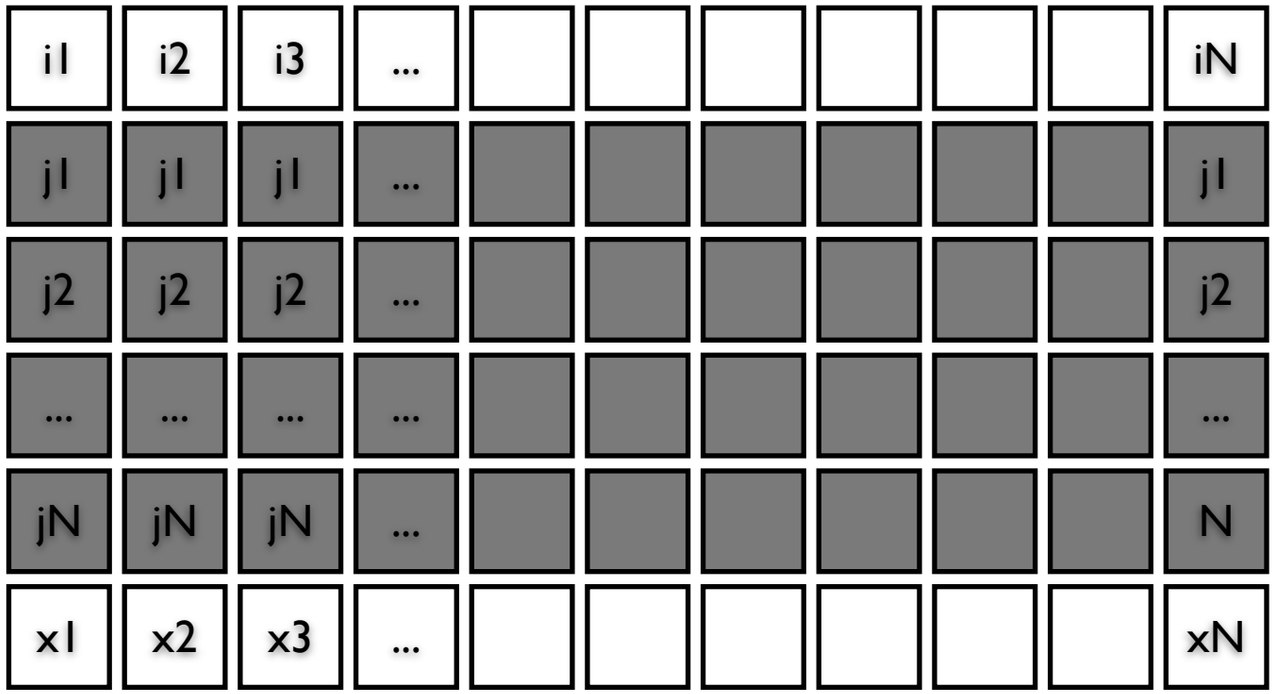
One interaction per thread

- Easy to implement
  - Reuse reduction algorithm
- Wasteful:
  - Need to store interactions before reduction
  - particle data is used once
  - need to repeat for each i-particle.



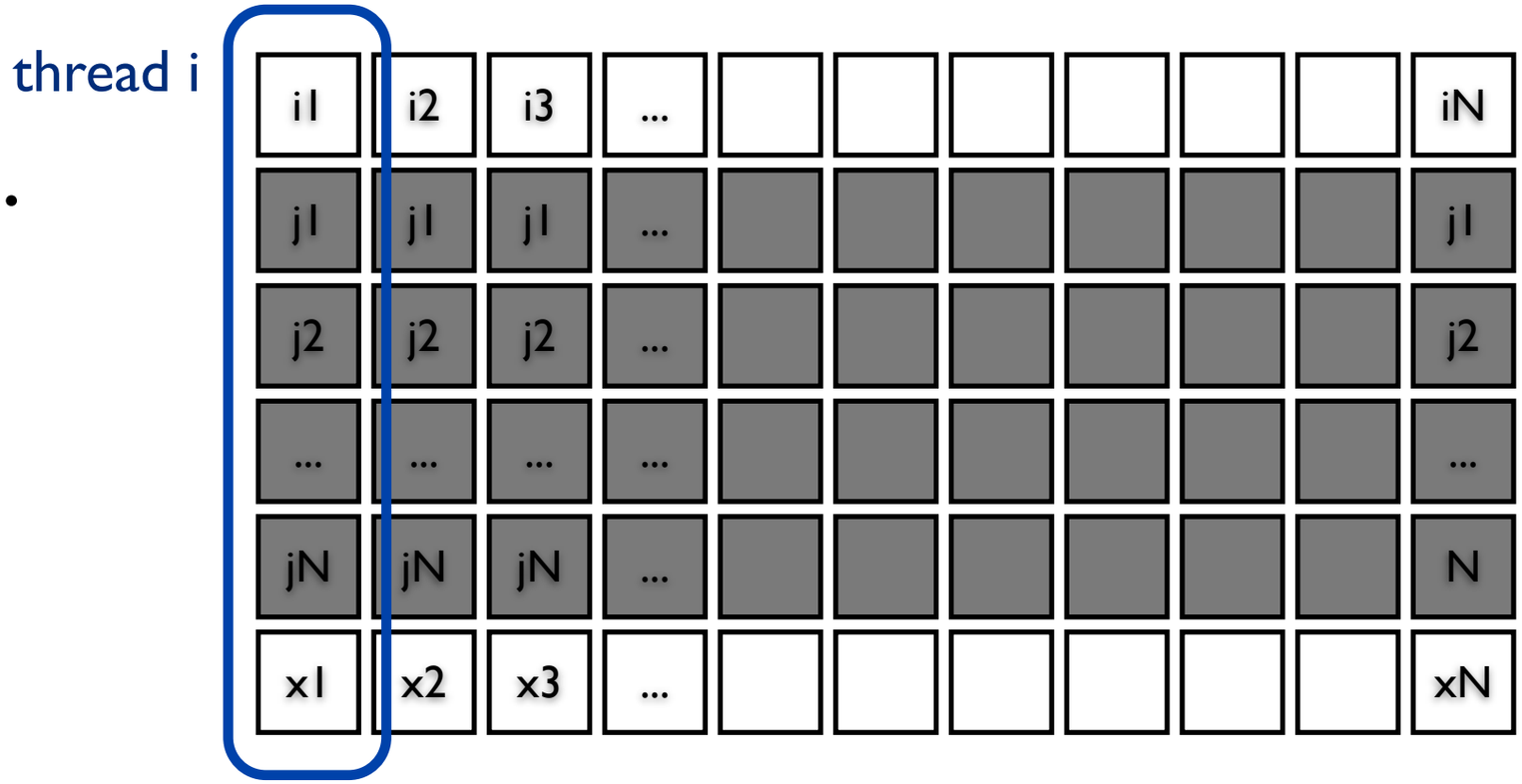
One i-particle per thread

- Uses few shared memory (or not at all).
  - for the i-particle data
- Does not need to reduce separately
- It is a long summation, might require Kahan's algorithm
- Wastes memory bandwidth, only one j accessed at any given time.



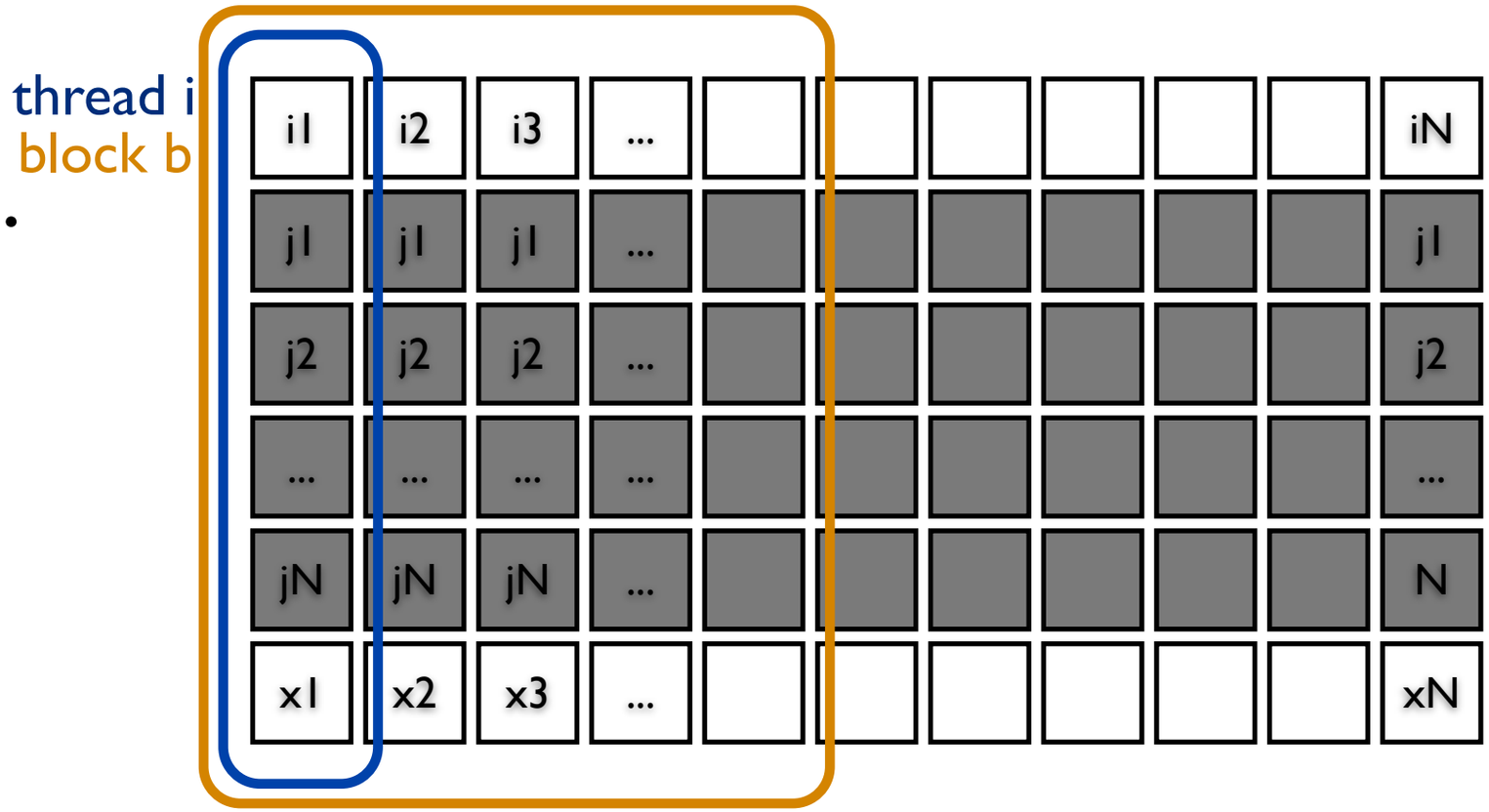
One i-particle per thread

- Uses few shared memory (or not at all).
  - for the i-particle data
- Does not need to reduce separately
- It is a long summation, might require Kahan's algorithm
- Wastes memory bandwidth, only one j accessed at any given time.



One i-particle per thread

- Uses few shared memory (or not at all).
  - for the i-particle data
- Does not need to reduce separately
- It is a long summation, might require Kahan's algorithm
- Wastes memory bandwidth, only one j accessed at any given time.



Multiple j-particle per block

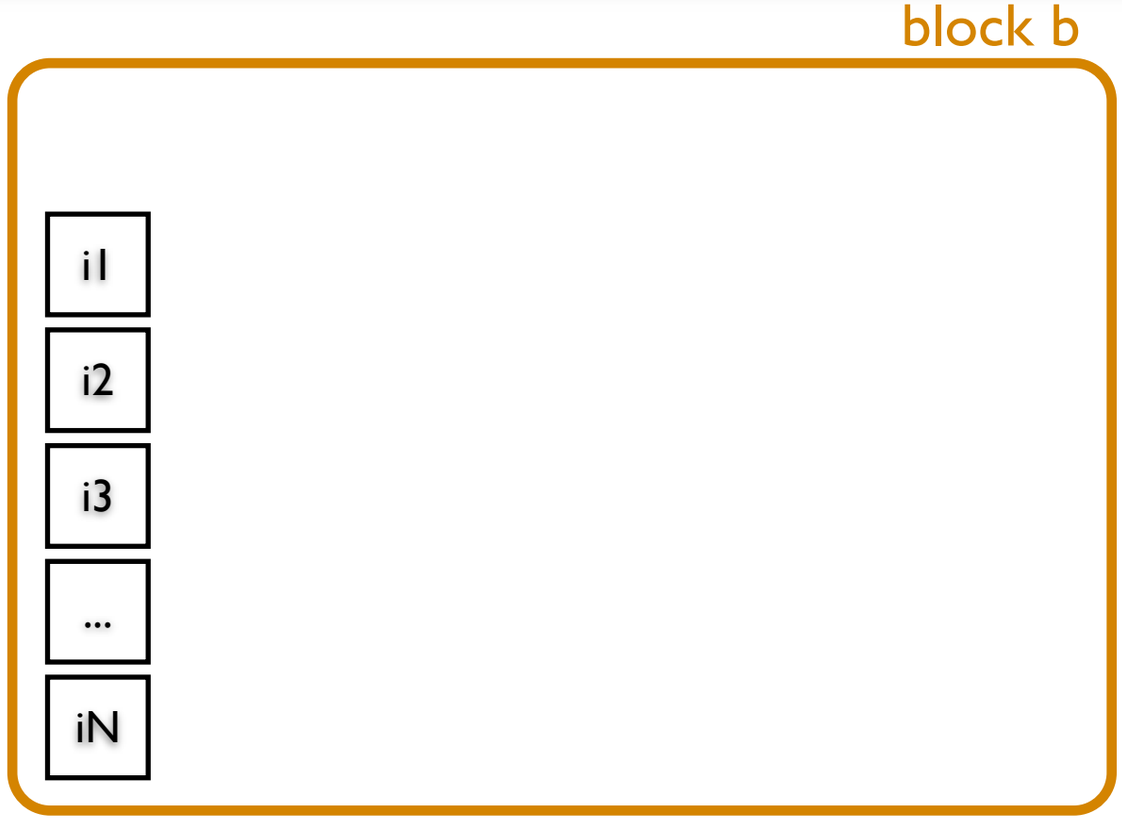
- Catches i and j particles per block
- Fij is computed but not stored
- Still wastes some bandwidth, because all blocks are accessing the same j-p range at the same time

Multiple j-particle per block

- Catches i and j particles per block
- Fij is computed but not stored
- Still wastes some bandwidth, because all blocks are accessing the same j-p range at the same time

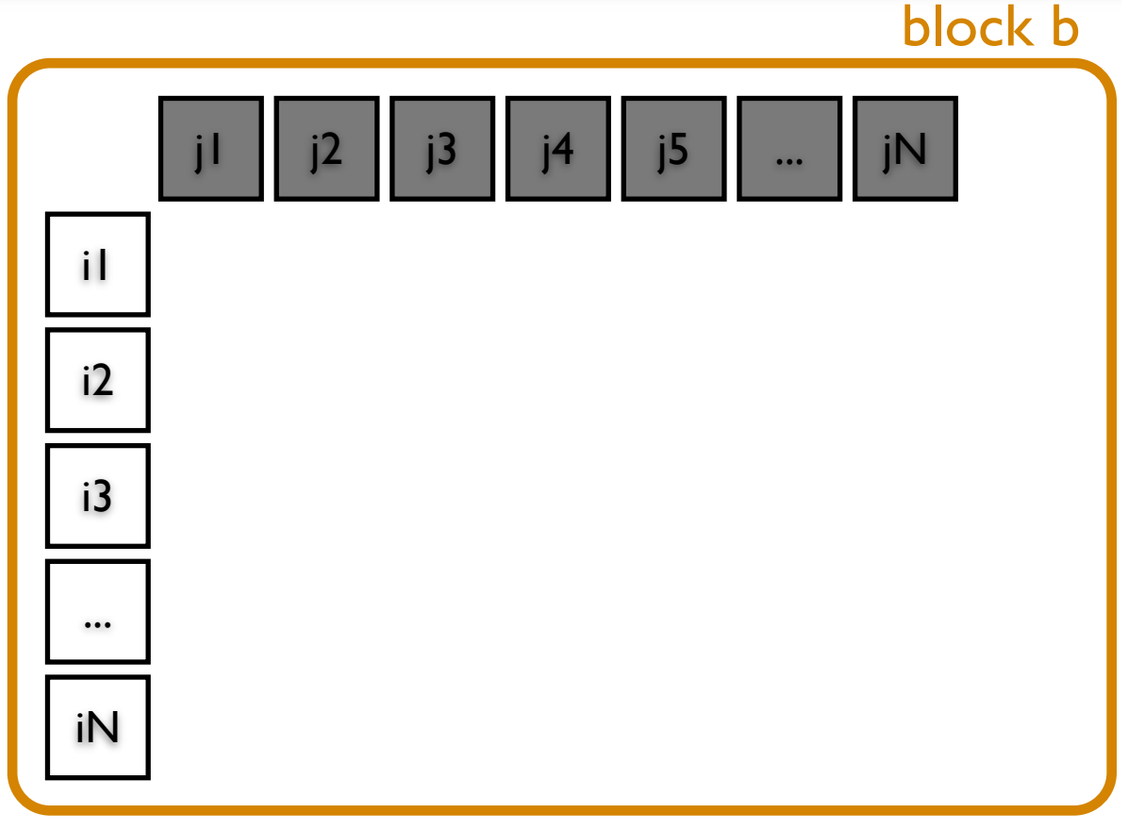
Multiple j-particle per block

- Catches i and j particles per block
- Fij is computed but not stored
- Still wastes some bandwidth, because all blocks are accessing the same j-p range at the same time



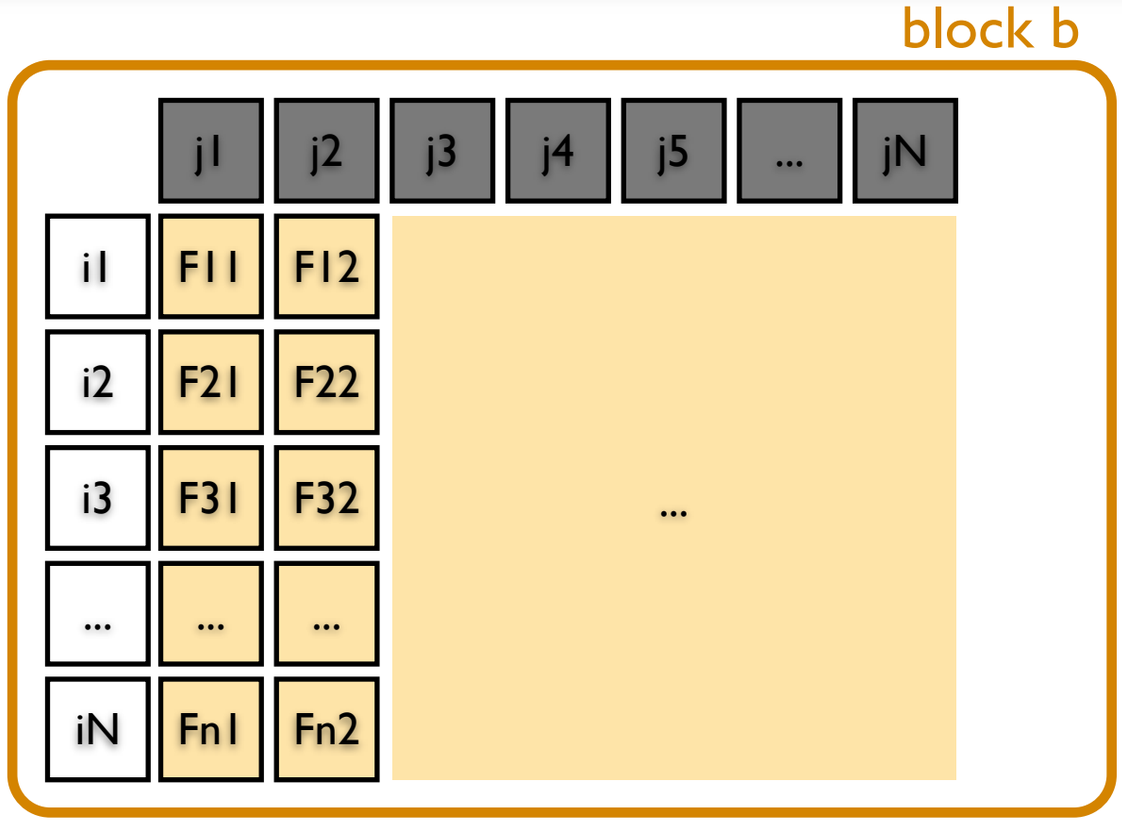
Multiple j-particle per block

- Catches i and j particles per block
- Fij is computed but not stored
- Still wastes some bandwidth, because all blocks are accessing the same j-p range at the same time



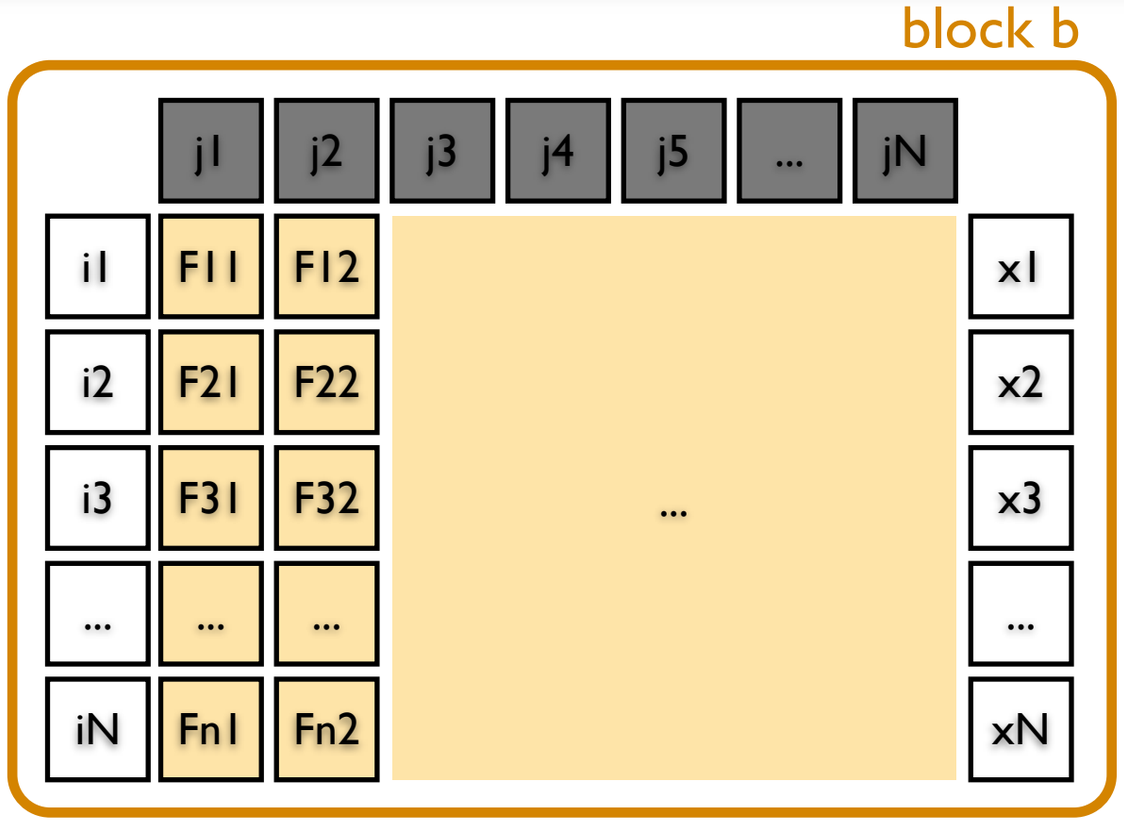
Multiple j-particle per block

- Catches i and j particles per block
- Fij is computed but not stored
- Still wastes some bandwidth, because all blocks are accessing the same j-p range at the same time



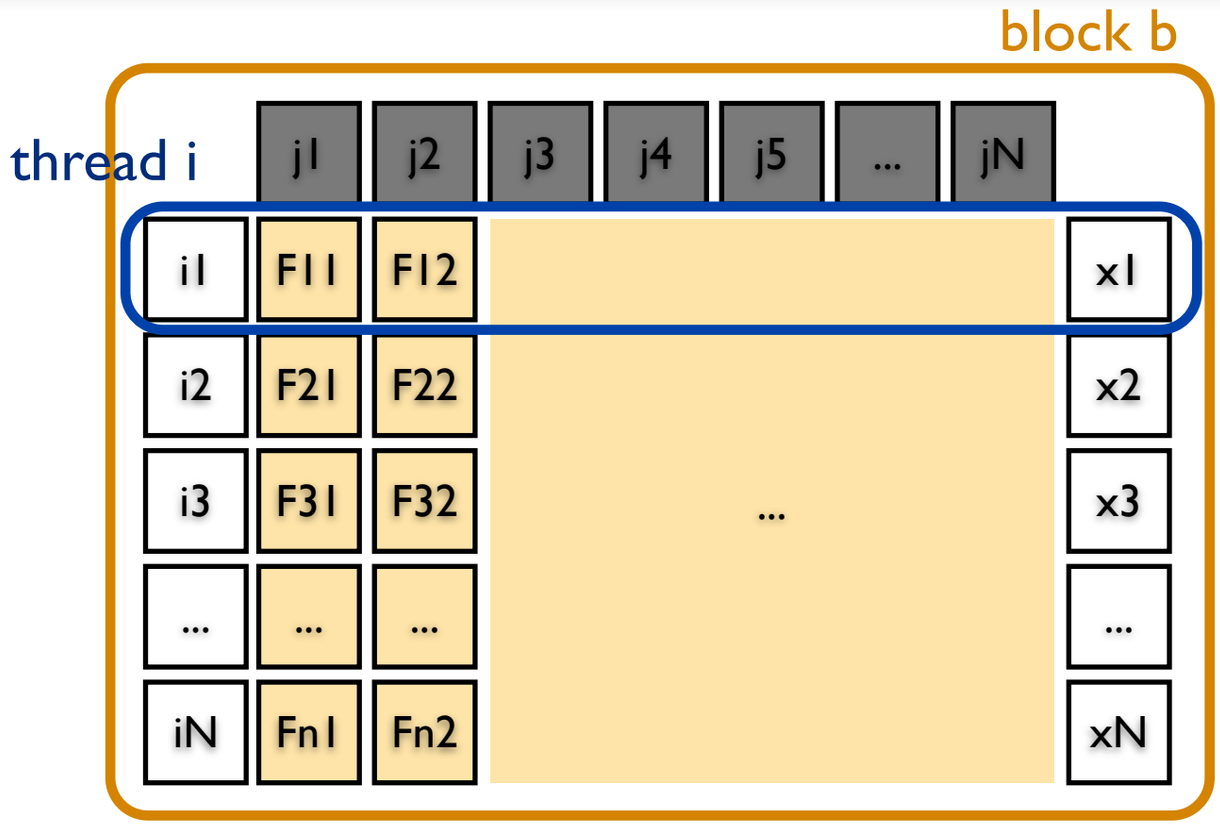
Multiple j-particle per block

- Catches i and j particles per block
- Fij is computed but not stored
- Still wastes some bandwidth, because all blocks are accessing the same j-p range at the same time



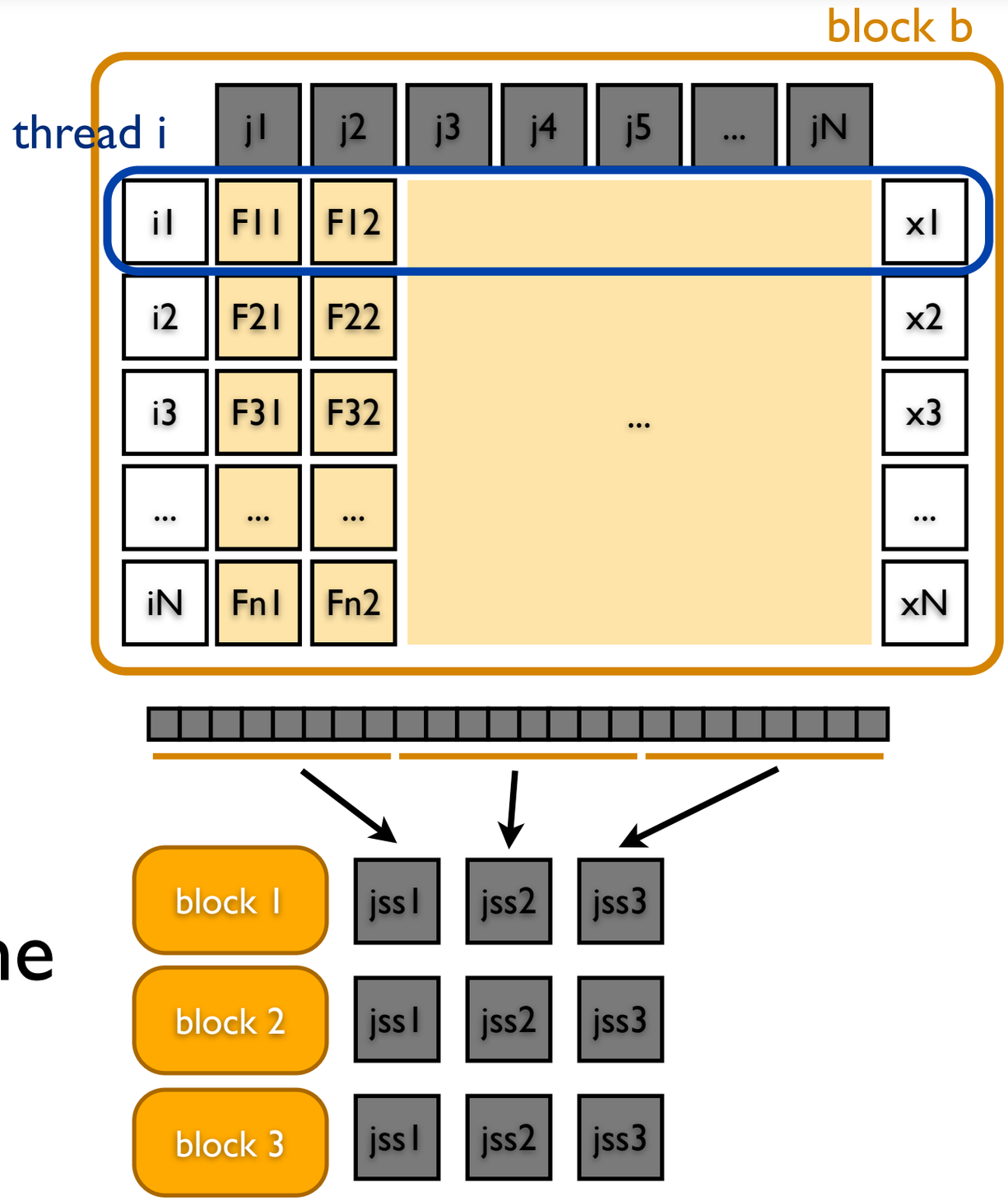
Multiple j-particle per block

- Catches i and j particles per block
- Fij is computed but not stored
- Still wastes some bandwidth, because all blocks are accessing the same j-p range at the same time

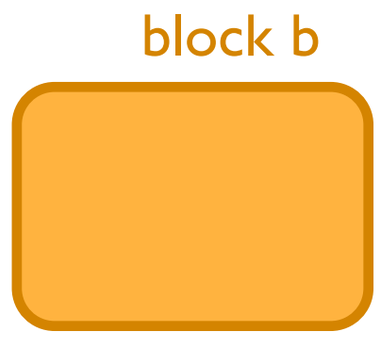


Multiple j-particle per block

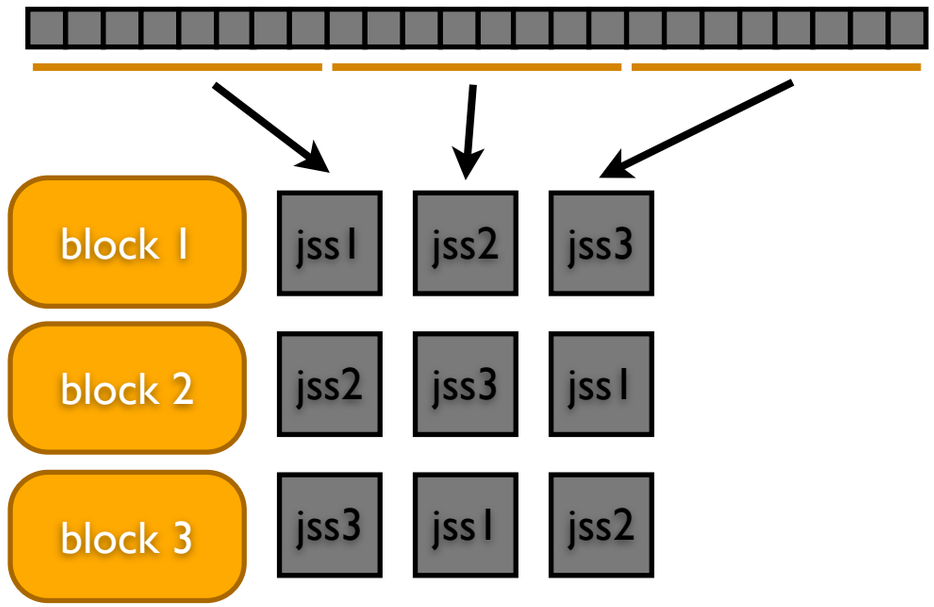
- Catches i and j particles per block
- Fij is computed but not stored
- Still wastes some bandwidth, because all blocks are accessing the same j-p range at the same time



multiple j-particle per block



- Every i-subset must process every j-subset
- Reorder the subset processing to improve the memory bandwidth



NBody Kernel code

```
#define ITEMS_PER_GROUP 64
#define SOFTENING 0.001

kernel void nbody_iterate(global float *oldPos, global float *newPos, global
float *vel, float delta_t,unsigned int numBodies) {

/* get ids */
    size_t gid = get_global_id(0);
    size_t lid = get_local_id(0);
    size_t gsz = get_global_size(0);


    float mass = 1.f / numBodies;            // total mass normalized to 1.
    float4 this_acc = (float4)0.f;

/* i-particles in local memory */
    local float4 ip[ ITEMS_PER_GROUP ];

/* fetch my i-particle */
    ip[lid].x = oldPos[4*gid+0];
    ip[lid].y = oldPos[4*gid+1];
    ip[lid].z = oldPos[4*gid+2];
    ip[lid].w = 0.f;
    mem_fence( CLK_LOCAL_MEM_FENCE );

/* iterate over j-particles */
    for (size_t j = 0; j < gsz ; j++)
    {
            if ( j < gsz ) {
                    float4 jp;
                    jp.x = oldPos[4*(j)+0];
                    jp.y = oldPos[4*(j)+1];
                    jp.z = oldPos[4*(j)+2];
                    jp.w = 0.f;
            float4 dist_vec = ip[lid] - jp;
                    float lps = length(dist_vec) + SOFTENING;
            this_acc += mass * dist_vec / (lps*lps*lps);
            }

    }
   barrier( CLK_LOCAL_MEM_FENCE );
```

```
/* compute velocity */
   float4 velocity;
   velocity.x = vel[4*gid+0] - (this_acc.x * delta_t);
   velocity.y = vel[4*gid+1] - (this_acc.y * delta_t);
   velocity.z = vel[4*gid+2] - (this_acc.z * delta_t);

/* simple “integration” */
   ip[lid] += velocity * delta_t;

/* save new velocity */
   vel[4*gid+0] = velocity.x;
   vel[4*gid+1] = velocity.y;
   vel[4*gid+2] = velocity.z;
   vel[4*gid+3] = 0.0f;

/* save new position */
   newPos[4*gid+0] = ip[lid].x;
   newPos[4*gid+1] = ip[lid].y;
   newPos[4*gid+2] = ip[lid].z;

   return;

}
```

Hands-on Session

- oclHelloVector - Hello Vector Example
  - Both C and C++ versions
- oclNBody - NBody example (non-OpenGL)

- Tasks to do in oclNBody:
  - Fetch the particle positions buffer from GPU, output to console
  - Modify the kernel, so each group works in a different subset of j-particles