User Manual
Version 4.1.1

Table of Contents

Chapter 1

Getting Started

1.1 Requirements

- C++ compiler compatible with C++-14 (recommended: C++-17)
- OS: The following operating systems are supported:
  - Linux (all distributions, glibc >= 2.18)
  - Windows
  - macOS >= 10.9
  - iOS >= 8.0
  - Android
- GPU: any modern graphics card compatible with OpenCL, CUDA, or Metal.
- CPU: all common CPUs are supported (current linux builds are available for x86_64, i686, arm64, arm32, power8)
- optional: cmake >= 3.3

1.2 Installation

Install scripts are provided for Linux and MacOS. Installation is optional, but recommended. Alternatively, the location of goopax can be specified to the build system by other means.

Linux

On linux, the script '`install.sh`' can be run as root to install goopax in the default system directories.

MacOS

On MacOS, the folder '`goopax.framework`' should be copied to '`/Library/Frameworks/`', or to '`$HOME/Library/Frameworks/`'.

Windows, iOS, Android

No installation scripts are provided for these operating systems. Simply move 'goopax-windows-4.1.1' to a custom location and set the paths appropriately (see below).

1.2.1 License File

The license file 'goopax_license.h' can be placed in the 'share/goopax/licenses' subfolder.

1.3 Building Programs

1.3.1 Building Programs with cmake

Cmake is the recommended way to build goopax programs. Simply use package goopax:

```
find_package(goopax)
```

And link the resulting target 'goopax::goopax' to your program:

```
target_link_libraries(my_program goopax::goopax)
```

If goopax is installed in a non-standard location, the path to <goopax>/share needs to be added to CMAKE_PREFIX_PATH.

If goopax_license.h is stored in a different location than 'share/goopax/licenses', the path to the folder where it is stored should be set in the 'GOOPAX_LICENSE_PATH' environment variable.

1.3.2 Building Programs without cmake

When using other build systems, it is up to the user to set the paths correctly. The compiler needs to find the goopax header files, as well as the license file 'goopax_license.h' in its search path. The linker needs to link to the goopax library.

1.4 Building the Example Programs

The example programs are a good place to start. Precompiled versions can be found in the 'bin', 'bin32', or 'bin64' folders. The source code is located in folder 'examples'. The examples can be built with cmake in the usual way, e.g.,

```
cd examples
mkdir build
cd build
cmake ..
cmake --build .
```

Chapter 2

Programming with Goopax

2.1 Program Structure

2.1.1 Including Header Files

To use Goopax, include the header file from your source code:

```
#include <goopax>
```

For OpenGL interoperability, include

```
#include <goopax_gl>
```

For OpenCL interoperability, include

```
#include <goopax_cl>
```

For Metal interoperability, include

```
#include <goopax_metal>
```

2.1.2 Namespaces

[sec:namespaces] The basic Goopax functionalities are found in namespace 'goopax'. For simplicity, we will assume that you import this namespace:

```
using namespace goopax;   // Basic GOOPAX types, such as buffer and gpu_float.
```

Optionally, one of the following namespaces can be used, for debug and release mode data types, respectively:

```
using namespace goopax::debug::types;    // Debugging data types Tint, Tfloat, ...
using namespace goopax::release::types;  // Release data types Tint, Tfloat, ...
```

For some functions that are provided by the C++ standard library, an overloaded function may be provided in the namespace "`std`".

2.2 Memory

In the GPU kernels, there are four types of memory:

- Private memory is not shared. Each thread has its own private memory.
- Local memory is shared between the threads in a work group. It is only valid during thread execution. Local memory can be used for thread communication within a work group, as well as sharing small data structures during kernel execution between the threads of a work group.
- Global memory is usually referred to as the main memory of a video card. It can be used to store the data used in calculations, and to share data between all threads, and between GPU and CPU. Global memory is accessible by all threads, and also from the CPU.
- lvm memory can be used across different devices and CPUs, within a single unified address space.

2.2.1 Private Memory

Private memory is allocated as follows:

```
private_mem<float> A(16);
```

This will allocate an array of 16 floats for each thread.

2.2.2 Local Memory

Local memory is only available during the execution of a kernel. Each work group has its own local memory. Local memory is useful for communication within a work group. Between work groups no sharing is possible.

Local memory is declared as `local_mem`<*type*>, which has the constructor

```
local_mem(size_t size)
```

For example:

```
local_mem<double> mem(256);
```

will allocate 256 doubles in local memory for each group.

2.2.3 Global Memory

Global memory must be declared on the CPU side as a `buffer` and on the GPU side as a `resource`.

Buffer

On the CPU side, the memory is declared as `buffer`<*type*>, where the element type is specified as a template parameter. The constructor takes the argument

```
buffer(goopax_device device, size_t size)
```

For example:

```
buffer<float> buf(default_device(), 10);
```

will allocate a global buffer on the video card of type `float` and size `global_size()`.

Resource

The resource<type> is declared from within a GPU kernel and has to match a corresponding buffer. Resources can either be declared as parameters, e.g.,

```
struct my_kernel :
  public kernel<my_kernel>
{
  void program(resource<int>& A)
  {
    <GPU code...>
  }
}
```

or it can be declared within the kernel body and linked to a specific memory buffer:

```
struct my_kernel :
  public kernel<my_kernel>
{
  buffer<float> Bb(1000);
  // <...>
  void program(...)
  {
    resource<float> Br(Bb);
  }
}
```

When the former constructor is used, the corresponding memory buffer has to be specified as an argument when calling the kernel. With the latter constructor, the resource can already be linked to a specific buffer, so that the buffer does not have to be supplied when executing the kernel.

2.2.4 Svm Memory

Svm memory is declared in a similar way as global memory, e.g.,

```
buffer<float> buf(default_device(), 10);
```

Instead of declaring a resource in the kernel, svm memory is accessed via a pointer. The pointer can be provided to the kernel via a normal parameter, e.g.,

```
void program(gpu_type<double*> ptr)
```

Not all devices support svm memory. Whether svm memory is supported can be checked with the function device::support_svm().

2.2.5 Memory Access

All memory types can be accessed within a kernel by the usual [] operator or by iterators.

2.2.6 Data Transfer from Host to GPU and from GPU to Host

Data transfer via DMA to and from the video card can be done with the following member functions of global memory buffer types.

copy_to_host(T* p)

Copies data from the buffer to the address in host memory specified by p.

copy_to_host(T* p, size_t beginpos, size_t endpos)

Copies data from the buffer to the address in host memory specified by p, in the range from position beginpos to endpos-1 within the buffer.

copy_from_host(T* p)

Copies data from the host address p to the buffer.

copy_from_host(T* p, size_t beginpos, size_t endpos)

Copies data from the host address p to the buffer, in the range from position beginpos to endpos-1 within the buffer.

2.2.7 Barriers

To avoid race conditions, it is sometimes necessary to place barriers between memory accesses, especially when local threads are communicating with each other. Race conditions can occur when two or more threads access the same memory address and their access is not properly synchronized.

Local barriers

Threads within a workgroup can be synchronized by calling the `local_barrier()` function, for example:

```
local_mem<float> a(local_size());
a[local_id()] = 2;
local_barrier();
gpu_float b = a[(local_id() + 5) % local_size()];
```

Without the barrier, a race condition would occur. Note: Local barriers only synchronize memory access within a work group. Memory between different groups is not synchronized.

Global Barriers

Global barriers can be placed to synchronize threads across workgroups:

```
resource<float> A;
resource<float> B;

A[global_id()] = 5;
B[2*global_id()] = 7;

global_barrier();   // This will place a barrier on all threads.

gpu_float x = A[0] + B[129];
```

2.3 User-Defined Types in Memory Access

In addition to using intrinsic types, memory access can also be done with user-defined classes. This can simplify code development and provide a more structured and safe way of accessing the data structures.

The following restrictions apply:

- The class must fully contain all the data of the data structure. Members that use external memory, such as `std::vector` are not allowed. However, one can use members of type `std::array`, or other classes that don't allocate memory.
- The data types used in the struct must be provided as template arguments (or being derived from them).
- Virtual functions or virtual base classes cannot be used.

To use user-defines types, additional information must be provided. In the general case, this requires providing specializations of the following structs:

- provide specialization of "`goopax::goopax_struct_type`" that returns the characteristic data type.
- provide specialization of "`goopax::goopax_struct_changetype`" in namespace "goopax" as a type change mechanism.

For structs where all template parameters are types used in the data structure, the predefined macro `GOOPAX_PREPARE_STRUCT` can be used instead. It must be used in the main scope, not within a namespace, and template arguments must be omitted.

GOOPAX_PREPARE_STRUCT

```
template <typename A, typename B> struct pair
{
  A first;
  B second;
};

// Easy case: Only types are used as template arguments. Can use the macro as follows:
GOOPAX_PREPARE_STRUCT(pair)
```

General case

```
template <typename T, size_t N>
struct array
{
    T data[N];
};

// Template arguments include non-types.
// Need to provide the following specializations by hand:
namespace goopax {
template<typename T, size_t N>
struct goopax_struct_type<array<T, N>>
{
    using type = T;
};
```

```
template<typename T, size_t N, typename X>
struct goopax_struct_changetype<array<T, N>, X>
{
    using type = array<typename goopax_struct_changetype<T, X>::type, N>;
};
}
```

2.3.1 Memory Access

Buffers and resources of the new type can be declared in the same way as buffers with intrinsic types, e.g.:

```
...
resource<complex<float>> my_global_resource;
local_mem<complex<float>> my_local_resource(local_size());
...

int main()
{
  ...
  buffer<complex<float>> my_buffer(default_device(), size);
}
```

The new type can be accessed in the usual C++ way. Some examples are shown here:

```
// assigning one item
my_global_resource[global_id()].real = 2.3;

// copying a complete element
my_local_resource[local_id()] = my_global_resource[12];

// modifying one item
my_global_resource[global_id()].imag += 25;
```

2.4 Images

Special data access is provided for images and image arrays. From the host code, images can be allocated with objects of type "`image_buffer`". From the device code, images can be accessed by using objects of type "`image_resource`". For more information, see the goopax reference.

2.5 GPU Kernels

Kernels are the functions that run on the video card and are typically used for the computationally demanding calculations.

2.5.1 Writing Kernel Classes, old syntax

GPU kernel classes are derived from the class template 'kernel', which takes the name of the user-defined kernel class as template argument. They should contain a function program(...) with the code which will be executed on the video card. A kernel might look like this:

```
class my_kernel :
  public kernel<my_kernel>
{
public:
  void program(<...resources...>)
```

```
  {
    <GPU code...>
  }
};
```

Kernel Instantiation

Before a kernel is first executed, it has to be instantiated, i.e., an object of the kernel class has to be created:

```
  my_kernel MyKernel;
```

The kernel can be declared as a static object. The kernel object is not bound to any specific device and can be executed on all available devices.

2.5.2 Writing Kernel Classes, new syntax

Starting from goopax 4.1.0, a new syntax for kernels is introduced. GPU kernel are specified as objects of type `goopax::kernel<arg_t>`, where `arg_t` is the function type. The device and the function with the kernel code are provided to the constructor.

2.5.3 Calling a Kernel

The kernel is executed by calling the '()' operator and passing the required buffers as arguments for all the unspecified resources. For example, if the kernel declares two resources

```
kernel<void(buffer<float>& A, const buffer<int>& B)>
my_kernel(default_device(),
          [](resource<float>& A, const resource<int>& B)
{
    ...
};
```

or, with C++17 template type deduction guides,

```
kernel
my_kernel(default_device(),
          [](resource<float>& A, const resource<int>& B)
{
    ...
};
```

then the kernel must be called with two buffers as arguments of the same type and in the same order, i.e.

```
buffer<float> A(default_device(), 100);
buffer<int> B(default_device(), global_size());
MyKernel(A, B);
```

All kernel calls are asynchronous. The call will, in most cases, return immediately. Only when the results are accessed, the CPU will wait for the kernel to finish.

2.5.4 Valid Kernel Arguments

Valid kernel arguments are buffers, input values, gather values, and images. The CPU argument type from the kernel call must correspond to the matching GPU argument type in the kernel definition:

CPU argument type	Kernel argument type
T	gpu_T
buffer<T>&	resource<T>&
const buffer<T>&	const resource<T>&
image_buffer<DIM, T>&	image_resource<DIM, T>&
image_array_buffer<DIM, T>&	image_array_resource<DIM, T>&
goopax_future<T>	gather_result<T>&

Here, T can be any intrinsic type or goopax struct. gpu_T is the corresponding GPU type.

2.5.5 Gathering Return Values

Return values can be combined from all threads, reduced into single values. The return type of the corresponding kernel argument is gather_result<T>, which has to be constructed from an appropriate reduction operator class. Pre-defined reduction operators are:

class	description
gather_add	calculate the sum of all values
gather_min	calculate the minimum of all values
gather_max	calculate the maximum of all values

Instead of using these pre-defined classes, user-defined classes can be used. For more information, see the example program "gather" or the header file "src/gather.h".

goopax_future

Gather values are wrapped into objects of type "goopax_future". Their behavior is similar to "std::future" of the standard C++ library. The actual values are returned by the "goopax_future::get()" function. If the return type of the kernel function is void, goopax_future<void> is returned when calling the kernel. Any goopax_future return object can be used to query information about the number of threads in the kernel, or the execution status.

Gathering values as references

When using references, multiple gather values can be used.
Example:

```
kernel testprog(device, [](gather_result<int>& minID, gather_result<int>& maxID)
{
    minID = gather_min(global_id());
    maxID = gather_max(global_id());
});

goopax_future<int> minID;
goopax_future<int> maxID;
testprog(minID, maxID);
cout << "minimum␣id=" << minID.get() << endl
     << "maximum␣id=" << maxID.get() << endl;
```

Gathering values as return values

Returning gathered values is done as shown in the following example:

```
kernel testprog(device, []() -> gather_result<float>
{
    return gather_add(global_id());
});

goopax_future<float> sum = testprog();
cout << "sum␣of␣thread␣IDs=" << sum.get() << endl;
```

2.6 Data Types

2.6.1 Basic GPU Data Types

Special data types are used to declare local variables that reside on the video card. The programming of GPU kernels relies on these data types. The following basic types are available:

GPU type	Corresponding CPU type
gpu_type<T>, gpu_T	T (T is an intrinsic type)
gpu_type<S*>	S* (S is an intrinsic type, or a user-defined struct)
gpu_type<const S*>	const S* (S is an intrinsic type, or a user-defined struct)

Valid intrinsic types are float, double, half, any integral type (signed or unsigned, 8, 16, 32, or 64 bits), or bool.

Variables of GPU type are used in the kernel function, or as local variables in other functions and classes that are called from the kernel function. They may only be used within the call stack of the kernel function.

2.6.2 CPU Types and GPU Types

In kernel development, we distinguish between variables of CPU type and of GPU type. This is not to be mistaken with the device type (section [sec:device]). Regardless of the device type, a CPU variable is an ordinary variable of the main program, whereas a GPU variable is a variable (typically a register) that resides on the device. Although kernel development relies on the use of GPU types, using CPU types within a kernel can sometimes simplify programming, and improve performance, as they are treated as constant expressions from the perspective of the GPU kernel.

Both CPU and GPU types can be used together in kernel programs. Instructions that use CPU types are calculated during kernel compilation. They are the equivalent of what constant expressions are in ordinary programs, and they will not use any GPU resources.

Hence, variables can be sorted into four different types of life cycle:

- **CPU-based compile-time** – Constant expressions that may be evaluated by the C++ compiler. Has no effect on the runtime of your program.
- **CPU-based run-time** – CPU variables evaluated at runtime. Has no effect on GPU kernels.

- **GPU-based compile-time** – GPU variables that can be evaluated by Goopax during kernel creation. May increase compilation time of the GPU kernels, but has no effect on the kernel runtime.
- **GPU-based run-time** These are the runtime variables that are used in kernel.

2.6.3 Matching CPU and GPU Types

It is sometimes necessary to get the corresponding CPU type from a GPU type or vice versa, especially when the type in question is a template parameter. This can be done with the structs goopax::make_cpu, and goopax::make_gpu. For a given input type T,

```
typename make_cpu<T>::type
```

is the CPU type, and

```
typename make_gpu<T>::type
```

is the GPU type. The type T can be a fundamental type, or a goopax struct.

2.6.4 Changing GPU/CPU Types

When writing template functions that should be valid both for CPU types and for GPU types, it is sometimes necessary to specify a type without knowing whether it will be used by the CPU or by the GPU.

The "change_gpu_mode" struct can generate a type regardless of whether it is for CPU or for GPU context. It takes a CPU type as template argument.

```
// T may be, e.g., float or gpu_float, or goopax::debug::types::Tfloat
template <class T> struct foo
{
  // If T is float, then D is double. If T is gpu_float, then D is gpu_double.
  using D = typename change_gpu_mode<double, T>::type;
};
```

2.6.5 Type Conversion Rules

Implicit Type Conversion

To avoid performance pitfalls, implicit type conversion is stricter than the usual type conversion of C++. Conversion will only be done automatically if the precision of the resulting type is at least as large as of the source type. Also, type conversion is done implicitly from integral types to floating point types, but not in the other direction from floating point types to integral types. No implicit type conversion is done from signed integral type to unsigned integral type.
Some examples of implicit type conversion:

```
gpu_float a = 2.5;       // ok, converting CPU double to GPU float.
gpu_double b = -3.7;
gpu_float c = b;         // Error: No implicit conversion from gpu_double to gpu_float.
gpu_float c = static_cast<gpu_float>(b);  // ok, explicit type conversion.
gpu_double d = a + b;    // ok, implicit conversion to type with higher precision.
```

Explicit Type Conversion

If the type conversion is not done implicitly, the type conversion can still be done explicitly in the usual C/C++ way, for example:

```
gpu_int64 a = 12345;
gpu_int b = (gpu_int)a;
gpu_int16 c = gpu_int16(b);
gpu_uint d = static_cast<gpu_uint>(b);
```

Type Conversion from CPU to GPU

Types are implicitly converted from CPU type to GPU type. The rules are slightly relaxed, as compared to GPU/GPU type conversion: CPU types can implicitly be converted to GPU types of lower precision. However, conversion from a CPU floating point type to a GPU integer type still requires explicit conversion.
Some examples:

```
int a = 5;
gpu_int b = a;
gpu_float c = 3.0;  // Conversion from double to gpu_float is ok.
gpu_int d = 3.0;    // Error: no implicit conversion from CPU floating point to GPU int.
```

Conversion in the other direction is not possible. A GPU type cannot be converted to a CPU type.

2.6.6 Reinterpret

Sometimes it is necessary to change the data type of a value without changing the binary content. The `reinterpret` function provides a general-purpose conversion mechanism. It is defined as:

```
template <class TO, class FROM>
  TO reinterpret(const FROM& from);
```

The source data is provided as a function parameter, and the destination type is provided as template argument. The "`reinterpret`" function can be used on various data types (GPU types, CPU types, user-defined GPU classes, pointers, buffers).
Some Examples:

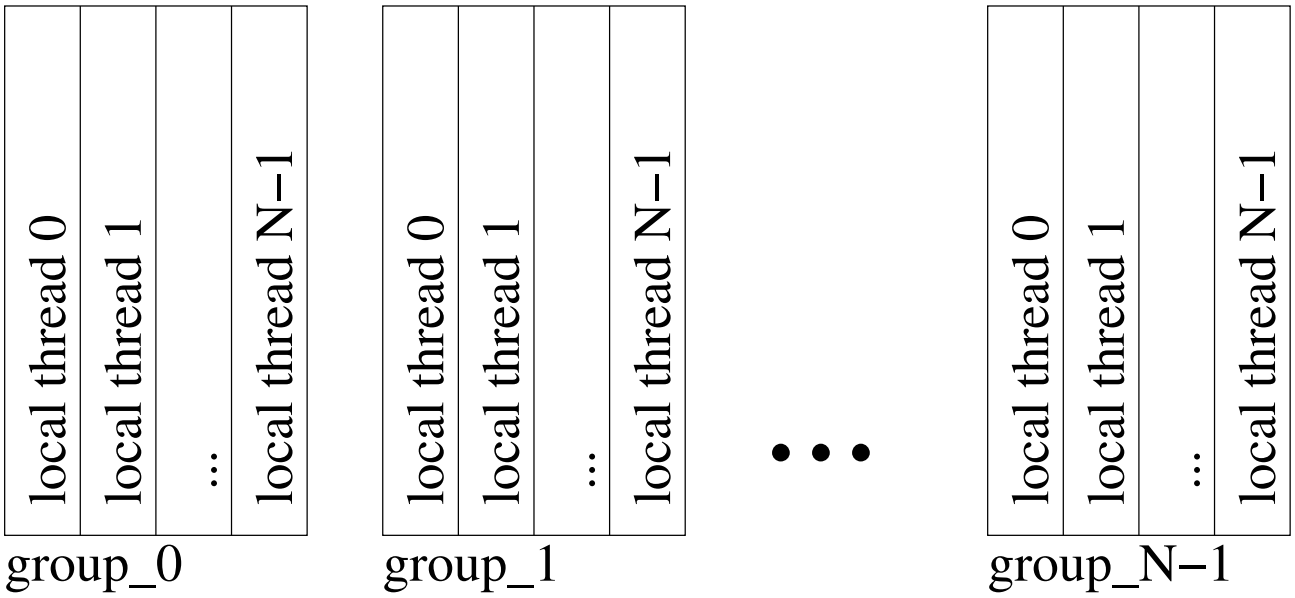
```
gpu_int a = 5;

gpu_float b = reinterpret<gpu_float>(a);       // Reinterpreting gpu_int to gpu_float

gpu_double c = reinterpret<gpu_double>(a);     // Error: different size!

array<gpu_int, 2> v = {{2, 3}};
gpu_double d = reinterpret<gpu_double>(v);     // Array of 2 gpu_ints to gpu_double.

int i = 2;
float f = reinterpret<float>(i);               // bit-casting a CPU value.

buffer<float> bf(device, 10);
buffer<int> bi = reinterpret<buffer<int>>(bf); // changing buffer value type

gpu_type<int*> p;
gpu_type<float*> pf = reinterpret<float*>(p);  // changing pointer type
```

2.7 Thread Model

The major difference between CPUs and GPUs is the level of parallelism. While CPUs are designed for serial computation, GPUs work vastly parallel, with thousands of threads working in parallel on a common goal.

2.7.1 Thread Numbers

Kernels are executed by all GPU threads in parallel. Each thread can query its ID by the functions `local_id()`, `group_id()`, and `global_id()` as described below. The threads are organized into several work groups, where each work group in turn consists of several local threads. Depending on the hardware, the threads in a work group may or may not work in unison, executing SIMD instructions that apply to all threads in that group at once. In any case, threads within a work group have better means of communicating with each other than threads that are in different work groups.



The number of local threads per group is given by `local_size()`. The number of groups is given by `num_groups()`. The total number of threads is therefore the product

```
global_size() = num_groups() * local_size().
```

Within a kernel, the local thread id can be queried by `local_id()`, the group id by `group_id()`, and the global id by `global_id()`. The global id is calculated as

```
global_id() == group_id() * local_size() + local_id().
```

For simple programs that can easily be parallelized, it may be sufficient to ignore the detailed thread model and to simply assume that the total number of threads is `global_size()`, and that each individual thread has the ID `global_id()`. For more complex programs, it may be beneficial to take the thread model into account and to separate the threads into work groups.

The group size and the number of groups can vary from one video card to another. The programmer should normally not make assumptions on these values, but always query them from `local_size()` and `num_groups()`. Although it is possible to manually set these values by the member functions "`force_local_size()`", "`force_num_groups()`", etc (see reference guide), one should normally not do this and let Goopax decide which thread sizes to use.

2.7.2 Threads of the same Work Group

Work groups are assumed to work in sync. They cannot branch off into different if-clauses or loops. Whenever only some threads of a work group enter an if clause, the other threads in the group must wait. The same is true for loops. To optimize performance, one should usually make sure that either all threads in a work group enter an if-clause or none, and that the number of loop executions is similar or equal for all threads in the group in order to avoid waiting times.

Threads of the same work group can communicate with each other via local memory or global memory.

2.7.3 Different Thread Groups

Different work groups can branch off into different if-clauses or for-loops. However, they cannot easily communicate with each other within one kernel execution. Atomic operations (section [sec:atomic]) offer some means of communication between different work groups.

Another possibility for communication between threads in different work-groups is to wait until the kernel execution has finished. Between two kernel calls it is guaranteed that all memory resources are synchronized.

2.8 Flow Control

2.8.1 Conditional Function

Simple if-statements can be expressed by a conditional function (the equivalent of the C/C++ conditional operator "a ? b : c"):

```
  T cond(gpu_bool condition, T return value if true, T return value if false)
```

2.8.2 If Clauses

For more complex statements that cannot be expressed as a conditional move, `gpu_if` can be used instead. Example:

```
gpu_if (local_id() == 0)
{
  a = 17;
}
```

2.8.3 Loops

gpu_for

Usage:

```
gpu_for<typename comp_t=std::less<>>(begin, end [,step], <lambda>);
```

Examples:

```
gpu_for(0, 10, [&](gpu_uint i)  // Counts from 0 to 9.
{
  ...
});

gpu_for<std::less_equal<>>(gpu_uint i=0, 10, 2)  // Counts from 0 to 10 with step 2.
{
  ...
});
```

gpu_for_global

The gpu_for_global loop is parallelized over all threads. It is provided for convenience.

```
gpu_for_global(0, N, [&](gpu_uint i)
{
  ...
});
```

is equivalent to

```
gpu_for(global_id(), N, global_size(), [&](gpu_uint i)
{
  ...
});
```

gpu_for_local

The gpu_for_local loop is parallelized over all threads in a work-group.

```
gpu_for_local(0, N, [&](gpu_uint i)
{
  ...
});
```

is equivalent to

```
gpu_for(local_id(), N, local_size(), [&](gpu_uint i)
{
  ...
});
```

gpu_break

Breaks out of a loop. It is called as a function of the loop variable. Usage example:

```
gpu_for(0, 10, [&](gpu_int i)
{
  gpu_if(i==5)
  {
    i.gpu_break();
  }
});
```

It is also possible to break out of a two-dimensional loop:

```
gpu_for(0, 10, [&](gpu_uint a)
{
  gpu_for(0, 10, [&](gpu_uint b)
  {
    gpu_if(a + b == 15)
    {
```

```
      a.gpu_break();   // Break out of both loops.
    }
  });
});
```

C-Style Loops

Traditional `for`-loops can also be used. They require that all loop boundaries are CPU values and therefore known to Goopax. The advantage is that the loop variable is also a CPU value, so that it can be used to address elements in `vectors` or `arrays`. In the following example, the sum of all elements of a 3D vector is calculated. The loop will be explicitly unrolled and all the calculation is done in registers.

```
std::array<gpu_float, 3> x = {{1,2,3}};    // A 3D vector

gpu_float sum = 0;
for (int k=0; k<3; ++k)
{
  sum += x[k];
}
```

Or, equivalently,

```
gpu_float sum = std::accumulate(x.begin(), x.end(), 0);
```

Warning: Using traditional C-style loops results in explicit loop unrolling. They should only be used in GPU kernels when the number of loop cycles are reasonably low!

2.9 Atomic Operations

Atomic memory operations are guaranteed to be indivisable and thread-safe. For example, if two threads atomically increase the value at the same memory location by 1, then the value will be increased by 2 in total, without the danger of getting into a race condition.

Atomic memory operations are supported on both global and local memory. The referenced memory objects must be 32 bit integers, or 64 bit integers. Atomic operations on 64 bit integers is not supported on all devices.

Example:

```
void program(resource<Tuint>& a)
{
  // Each thread adds 5 to a[0].
  gpu_uint k = atomic_add(a[0], 5);
}
```

2.9.1 Atomic Functions

The following atomic functions are supported.

In these functions, **REF** is a reference to a location in global or local memory, and may reference to a value within a user-defined struct. **T** is the value type of the memory and must be one of "`gpu_int`", "`gpu_uint`", "`gpu_int64`", "`gpu_uint64`".

T atomic_add(REF, T value)

Adds *value* to the memory location referenced by *REF*. The original value is returned.

T atomic_sub(REF, T value)

Subtracts *value* to the memory location referenced by *REF*. The original value is returned.

T atomic_min(REF, T value)

Calculates the minimum value. The original value is returned.

T atomic_max(REF, T value)

Calculates the maximum value. The original value is returned.

T atomic_and(REF, T value)

Calculates bitwise and. The original value is returned.

T atomic_or(REF, T value)

Calculates bitwise or. The original value is returned.

T atomic_xor(REF, T value)

Calculates bitwise xor. The original value is returned.

T atomic_xchg(REF, T value)

Exchanges *value* with the memory location referenced by *REF*. Returns the original value. *T* must be one of "gpu_int", "gpu_uint", "gpu_float", "gpu_int64", "gpu_uint64", or "gpu_double".

T atomic_cmpxchg(REF, T cmp, T value)

Compares *cmp* with the memory location referenced by *REF*. If both are the same, write *value* to the memory location. The original value is returned. *T* must be one of "gpu_int", "gpu_uint", "gpu_int64", "gpu_uint64".

2.10 General Programming Guidelines

Programming with Goopax is generally safe. Most programming errors are detected at compile-time, some are detected at program run-time. Others can be detected by the extensive error checking mechanisms. However, there are a few things one should keep in mind.

2.10.1 Using existing non-goopax libraries

Many existing template-libraries can be used with goopax, by using gpu types as template arguments. This is generally safe. If the code is compiled successfully, it will work. If the function cannot be used, because it contains if-clause or loops, the compiler will produce an error.

2.10.2 CPU loops and Recursive Calls

Using C/C++-style loops and recursive function calls can result in very fast kernel code. However, because the resulting code is completely unrolled, this can quickly use up the available registers. Only use CPU loops if the number of iterations is small.

2.10.3 Break, return, continue

Use caution when using these statements in kernels.

`break`, `continue`

Never use `break` or `continue` statements within an `gpu_if` clause, or within GPU loops. Use `gpu_break` or `gpu_continue` instead.

`return`

Never use the `return` statement within an `gpu_if` clause or within GPU loops. Always return values at the end of the function.

2.10.4 CPU/GPU interoperability

It is often useful to define functions and classes in such a way, that they can be used both from CPU code and from GPU code, depending on the type of the template parameters. This can result in more compact, more maintenable code. Simple functions may work without modifications. If the function contains if clauses or loops, additional modifications may have to be applied. To simplify programming, certain goopax statements can be used both in CPU code and GPU code:

`gpu_for`, `gpu_for_(global|local|group)`, `for_each`, `for_each_(global_local_group)`

If the loop variable is of CPU type, then these loops will behave as normal C++ loops and can be used in CPU code. The `_global`, `_local`, and `_group` loops are treated as their normal `gpu_for`

or `for_each` pendants, respectively.

`gpu_if`, `gpu_else`, `gpu_elseif`

If the conditional variable can be converted to `bool`, these statements will be treated as normal C/C++ if/else statements.

2.10.5 Number of Threads

The number of threads that are used in kernel execution are typically not specified by the programmer, but are chosen by goopax, depending on how many threads work best on the GPU. The number of threads may vary from kernel to kernel, depending on register use. Upper limits are provided by `global_size()`, `local_size()`, and `num_groups()`.

Chapter 3

Operators and Functions

All the usual C++ operators and math functions have been overloaded for the GPU types and may freely be used. Only the conditional operator ("`a ? b : c`") must be replaced by the conditional function `cond`.

3.1 Operators

3.1.1 Operators for Floating Point numbers and Integers

- Arithmetic: "+", "-", "*", "/"
- Increment/Decrement: "++", "--"
- Comparison: "==", "!=", ">", ">=", "<", "<="
- Assignment: "="
- Compound assignment operators: "+=", "-=", "*=", "/="

3.1.2 Operators for Integers

- shift operators:
  - "<<"
  - ">>"

  Left/right shift a by b bits. b must be smaller than the number of bits in a.
- "&" (binary and), "|" (binary or), "^" (binary xor)
- "%" (modulo operator)
- Compound assignment operators: "<<=", ">>=", "&=", "|=", "^=", "%=" %=

3.1.3 Boolean Operators

- && (boolean and)
- "||" (boolean or)
- "!" (boolean not)

3.2 Floating Point Functions

Mathematical functions on GPU types can be used in the same way as they are used on CPU types, for example:

```
gpu_float x = 0.5;
gpu_float s = exp(x);
```

Goopax will use the best implementation for the given video card, based on performance measurements. It will select between native functions, OpenCL implementations, and Goopax implementations.

3.2.1 Unary Functions

- cos, cospi, acos, acospi
- sin, sinpi, asin, asinpi
- tan, tanpi
- sinh, asinh
- cosh, acosh
- tanh
- exp, exp2
- log, log2
- sqrt, cbrt
- erf, erfc
- tgamma
- ceil, floor, round
- isfinite, isinf, isnan, isnormal

3.3 Integer Functions

T clz(T)

Returns the number of leading zeros.

gpu_uint popcount(T)

Counts the number of bits set to 1.

T rotl(T a, gpu_int bits)

Rotate left. *bits* can be any number, positive, negative, or zero.

T rotr(T a, gpu_int bits)

Rotate right. *bits* can be any number, positive, negative, or zero.

3.4 Functions for Integers and Floats

gpu_T min(gpu_T a, gpu_T b)

Returns the minimum value of a and b.

gpu_T max(gpu_T a, gpu_T b)

Returns the maximum value of a and b.

gpu_TU abs(gpu_T a)

Returns the absolute value of *a*. If a is a signed integer, the result is an unsigned integer.

3.5 Work-Group Functions

gpu_bool work_group_any(gpu_bool x)

Returns true if x is true for any thread in the work-group.

gpu_bool work_group_all(gpu_bool x)

Returns true if x is true for all threads in the work-group.

T work_group_reduce_add(T x)

Returns the sum of all values of x in the work-group.

T work_group_reduce_min(T x)

Returns the minimum value of x in the work-group.

T work_group_reduce_max(T x)

Returns the maximum value of x in the work-group.

T work_group_broadcast(T x, gpu_uint local_id)

Broadcasts value x to thread local_id in the work-group.

work_group_scan_inclusive_(T x)
work_group_scan_exclusive_(T x)

Does a prefix-sum operation over all threads in the work-group. may be sum, min, or max. Example:

local_id	0	1	2	3	4 ...
x	1	0	2	5	1
work_group_scan_inclusive_add	1	1	3	8	9
work_group_scan_exclusive_add	0	1	1	3	8

3.6 Random Numbers

Goopax provides a WELL512 random number generator. It is very fast – all the calculation is done in registers – at the expense that the random numbers should be used in blocks.

To use the random numbers, it is first necessary to create an object of type `WELL512`, which contains a buffer object to store the seed values. Then, in the GPU kernel program, an object of type `WELL512lib` should be created, which will provide the random numbers. The constructor of `WELL512lib` takes the `WELL512` object as input parameter.

The function `WELL512lib::gen_vec<`*type*`>()` returns a vector of random numbers of the specified type (which can be of floating point or integral type). The size of this vector depends on its type. It is 8 for 64-bit values, 16 for 32-bit values, and larger for smaller integer types. The best performance is achieved if all values in the vector are used before a new vector is generated. Floating point random numbers are in the range $0..1$, unsigned integers from 0 to the maximum value, integers from the largest negative value to the largest positive value.

See the following example:

```
struct random_example :
  kernel<random_example>
{
  WELL512 rnd;                  // Object for the seed values

  void program()
  {
```

```
    WELL512lib rnd(this->rnd);  // Instantiate the random number generator

    vector<gpu_float> rnd_values = rnd.gen_vec<float>(); // Generate random floats
    ...
  }
};
```

Chapter 4

Error Checking Mechanisms

4.1 Overview

Goopax offers extensive support for automatic error checking. This includes

- Checking for overflow of memory resources
- Checking that all variables are properly initialized
- Detecting race conditions

These error checking mechanisms can be enabled to look for bugs in the program. They are not intended for use in release mode, because they cannot be run on the video card, and they also reduce performance for CPU mode.

4.2 Enabling Error Detection Mechanisms

For error detection mechanisms, special debug data types are provided in the namespace goopax::debug::types. Debug types are prefixed by "T", e.g., "`Tfloat`", "`Tuint64_t`", or "`Tbool`". In difference to the corresponding intrinsic types, debug types will detect and report the use of uninitialized variables. If used in buffers, race conditions will be detected.

4.2.1 Using the Debug Namespace

The proposed way is to import the namespace `goopax::debug::types` in debug mode and the namespace `goopax::release::types` in release mode.

Your program could start like this:

```
#include <goopax>

using namespace goopax;
#if USE_DEBUG_MODE
using namespace goopax::debug::types;
#else
using namespace goopax::release::types;
#endif
```

Here, the debug mode is enabled by the compiler switch "USE_DEBUG_MODE".

4.2.2 Enabling Error Checking in Kernels

To enable error checks in the kernels, all `buffer`, `resource`, and `local_mem` types should use `T`-prefixed types, like this:

```
struct my_kernel :
  kernel<my_kernel>
{
  buffer<Tdouble> A;

  void program()
  {
    resource<Tdouble> A(this->A);
    local_mem<Tint> B(2*local_size());
    ...
  }
  ...
};
...
```

This will enable all error checks in the kernels, if the data types from the debug namespace are used.

4.2.3 Extending the Error Checks to the CPU Program

Error checking mechanism can also be used in the CPU program, by using the T-prefixed data types throughout the program, instead of intrinsic c++ data types. This will provide extensive checks for variable initialization.

This should work in most cases. However, be aware that this may cause compilation errors from time to time (for example, the `main` function requires plain C++ types, as do constant expressions). Such errors have to be resolved by explicit type conversions or by reverting to the C++ intrinsic types.

Warning: Be cautious about `sizeof`! The sizes of debug variables are larger than the sizes of the original data types. Use

```
gettype<}{\it type}}>::size
```

instead, it will return the size of the original, intrinsic C++ type, and can be used on GPU types as well:

```
sizeof(int)             // returns 4
sizeof(Tint)            // undefined, don't use!
sizeof(gpu_int)         // undefined, don't use!
gettype<Tint>::size     // returns 4
gettype<gpu_int>::size // returns 4
```

4.3 Running Programs in Debug Mode

The debug mode is only available on the CPU device. The CPU device can be selected with `env_CPU` when calling the functions `default_device` or `devices`. By default, the number of threads is equal to the number of available CPU cores, and the number of groups is 1. The number of threads can be

changed by calling the `force_local_size` and `force_num_group` member functions of the device. This may be helpful to mimic the behavior of the video card.

4.4 Debugging Errors

When an error is detected, an appropriate error message is generated and the program is terminated. A debugger can be used to pin down the point where the error occurs.

To do this, it is helpful to disable optimization and to enable debugging symbols by passing appropriate compiler options with the `GOOPAX_CXXADD` environment variable and to disable coroutines with `GOOPAX_COROUTINES=0`.
Example:

```
GOOPAX_CXXADD='-O0 -ggdb3' GOOPAX_COROUTINES=0 gdb ./my_program
```

Chapter 5

OpenCL Interoperability

Goopax can be used in conjunction with existing OpenCL code. For this to work, the same OpenCL platform, context, and device must be used in Goopax and in your OpenCL code, and the same OpenCL queue should be shared between OpenCL and Goopax. Memory buffers can then be shared between Goopax and OpenCL.

To use OpenCL Interoperability, the header file <goopax_cl> must be included:

```
#include <goopax_cl>
```

To see how OpenCL interoperability is applied, also see the example programs "cl_interop_1" and "cl_interop-2".

5.1 Accessing Goopax Resources from OpenCL

The following functions provide access to Goopax resources from OpenCL code.

Platform:

```
cl_platform_id get_cl_platform()
```

Returns the OpenCL platform that is used by Goopax.

Context:

```
cl_context get_cl_context()
cl::Context get_cl_cxx_context()
```

Returns the OpenCL context that is used by Goopax.

Device:

```
cl_device_id get_cl_device()
```

Returns the OpenCL device that is used by Goopax.

Buffers and Images:

```
template <class BUF> inline cl_mem get_cl_mem(const BUF& buf)
template <class BUF> inline cl::Buffer get_cl_cxx_buf(const BUF& buf)
```

Returns the OpenCL memory handle for the Goopax buffer or image.

Chapter 6

OpenGL Interoperability

Goopax can share memory resources with OpenGL.

6.1 OpenGL Initialization

To enable OpenGL support, the goopax device must be retrieved from the `get_device_from_gl` function.

6.2 Sharing Buffers and Images with OpenGL

Goopax images and buffers can be created from existing OpenGL objects by using the static member functions "`create_from_gl`".

```
buffer::create_from_gl(goopax_device device, GLuint GLres, uint64_t cl_flags=CL_MEM_READ_WRITE)
  image_buffer::create_from_gl(GLuint GLres, uint64_t cl_flag=CL_MEM_READ_WRITE,
                               GLuint GLtarget=GL_TEXTURE_2D, GLint miplevel=0)
```

GLres: The OpenGL object ID.

cl_flags: The OpenCL access mode.

GLtarget: The OpenGL object type.

GLint: The OpenGL miplevel

For example, if "`gl_id`" is the ID of a 2-dimensional OpenGL texture with 32 bit data size, then

```
  image_buffer<2, uint32_t> A = image_buffer<2, uint32_t>::create_from_gl(gl_id)
```

creates an image "`A`" that can be used with Goopax.

Chapter 7

Example Programs

The source code of these programs is included in the goopax package.

7.1 pi

This program approximates the value of π in a very simple way: It uses a WELL512 random number generator to produce points in a 2-dimensional space $0 < x < 1$ and $0 < y < 1$ and counts, how many of those points lie within a circle of radius 1 from the origin, i.e. $x^2 + y^2 < 1$. This fraction multiplied by 4 approximates the value π. This program supports MPI and can be run with the `mpirun` command to combine the computing power of several hosts or video cards.

7.2 Mandelbrot

This program calculates a Mandelbrot image and uses OpenGL to draw it on the screen. It is an interactive program that can be controlled with the left mouse button and the forward and backward arrow keys.

7.3 Deep Zoom Mandelbrot

This is a somewhat more complex Mandelbrot program, combining arbitrary-precision arithmetics on the CPU with a special algorithm of our design. It allows to zoom in to large magnification levels (factor 10^{80} and more), and still display the result in real time.

7.4 fft

This program applies Fourier transforms on live camera images, and filters out high frequencies or low frequencies.

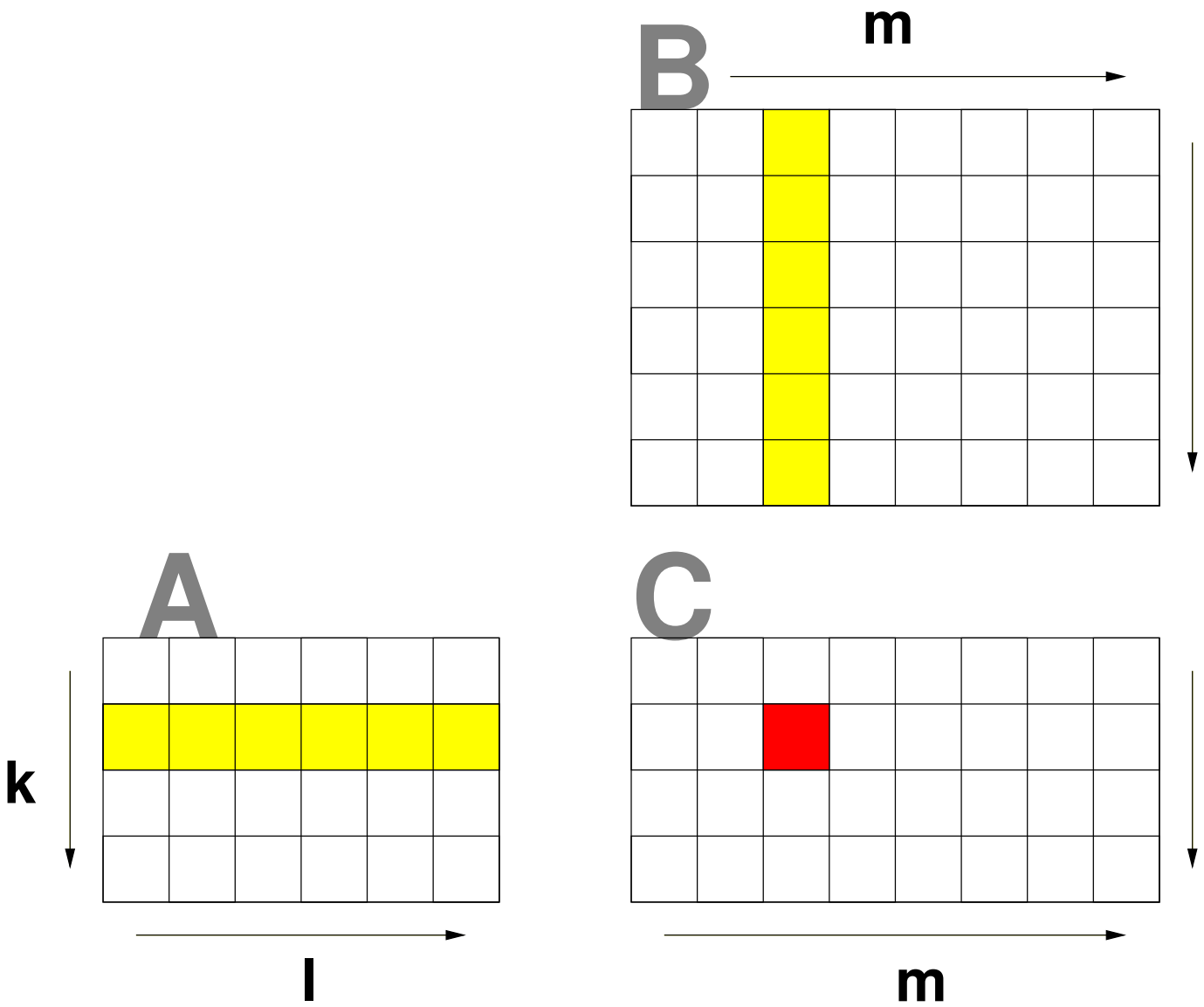
7.5 nbody

N-body simulation. Two colliding galaxies are simulated, each consisting of particles. Each particle interacts via gravity with all other particles. The particles are displayed via OpenGL.

7.6 matmul

The example program '`matmul`' performs matrix multiplications. It uses the naïve multiplication algorithm. For productive purposes, faster algorithms such as Strassen's algorithm should be considered as well. The major performance bottleneck is the memory access. Three different algorithms are implemented that use different techniques to reduce the access to global memory.

7.6.1 Naïve Implementation

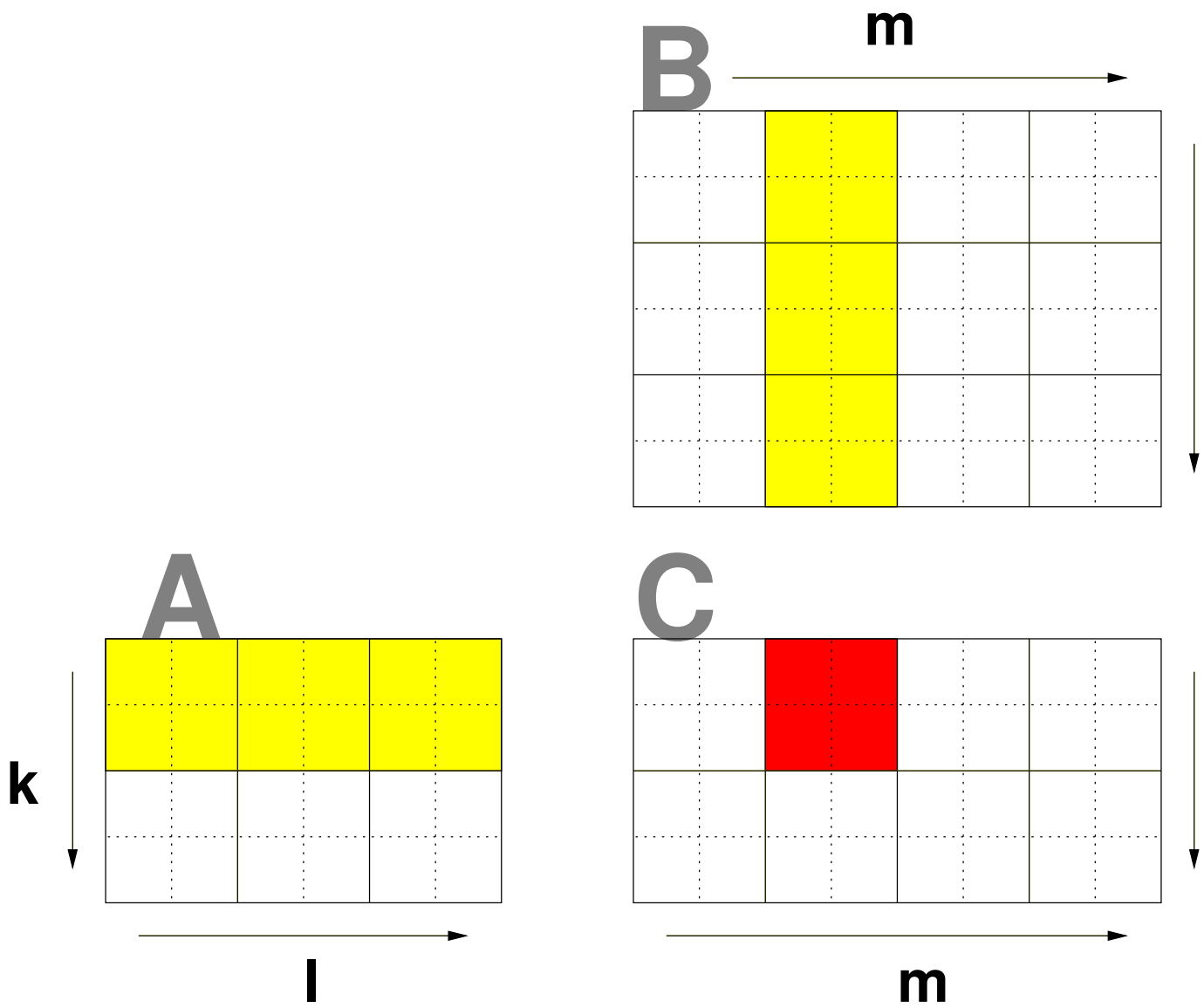


In the naïve implementation, a 2-dimensional loop is executed over all matrix elements in the result matrix C, with indices k and m. This loop is parallelized over all threads, so that every element in

C is handled by exactly one thread. For every matrix element, the value $C[k][m]$ is calculated as a dot product of the k'th row of matrix A and the m'th column of matrix B.

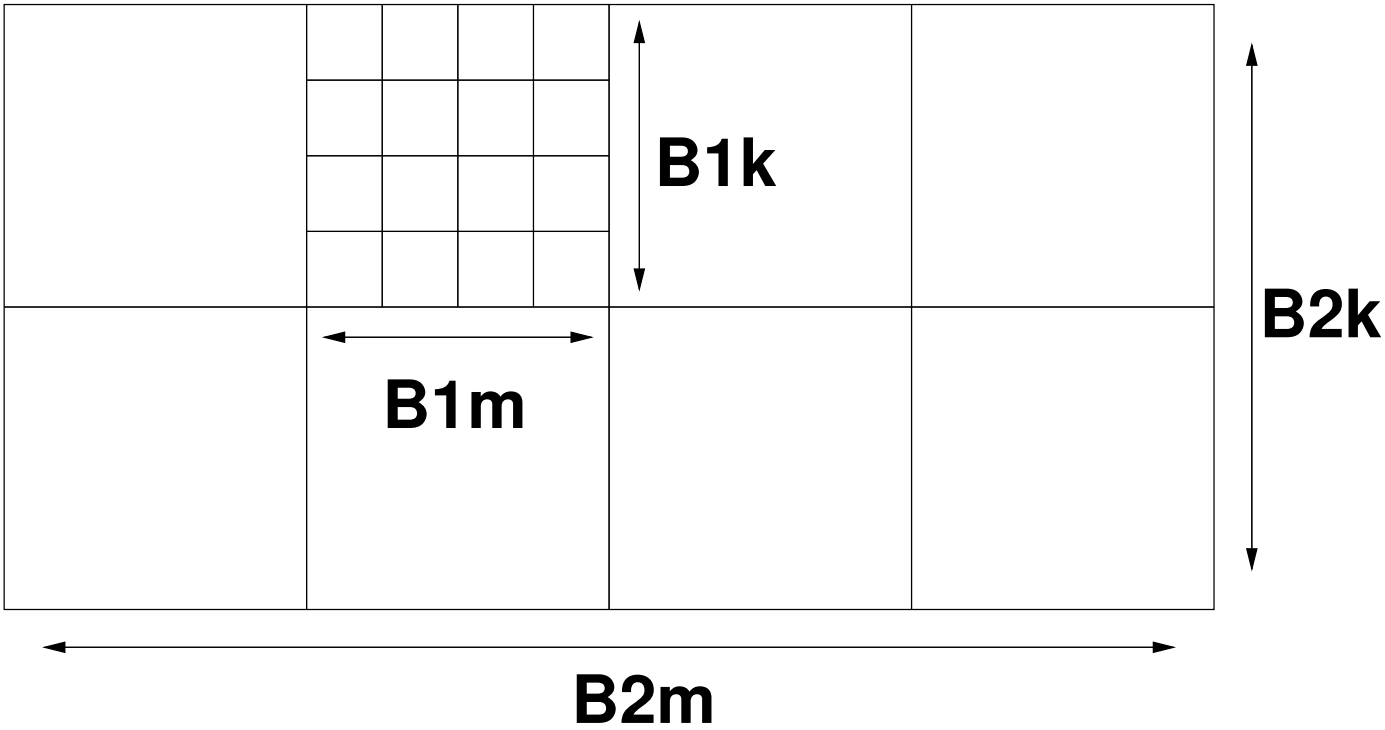
The problem with this naïve implementation is the frequent access to global memory.

7.6.2 Caching sub-Blocks in Registers



In the `matmul_reg` implementation, the matrix C is divided into sub-matrices of block size `B1m` x `B1k`. Each block in the result matrix C is handled by exactly one thread. For the calculation of the sub-blocks of matrix C, the values of the corresponding blocks of matrices A and B are stored in registers. This reduces the number of memory accesses to matrix A by a factor `B1m` and to matrix B by a factor `B1k`.

7.6.3 Caching sub-Blocks in Registers and Local Memory



The `matmul_reg_and_localmem` implementation takes the tiling one step further and uses both registers and local memory. Result matrix C is divided into large blocks of size $(B1m \times B2m) \times (B1k \times B2k)$. Each large block is handled by one work-group. This large block is sub-divided into small blocks of size $B1m \times B1k$, one block for every thread in the work-group. Each thread then computes its small block, using registers for the small block calculation, and local memory to fetch the data from within the large block.