http://www.multicorewareinc.com

GMAC User Manual

Multicoreware Inc.

June 8, 2011

Revision History

Date	Author	Changes
May, 2011	Isaac Gelado and Javier Cabezas	Initial version of the manual

Contents

List of Figures

Listings

Chapter 1

The GMAC Programming Model

This chapter provides a general description of the GMAC programming model. It explains the GMAC asymmetric distributed shared memory model and the two different threading models provided by GMAC.

1.1 The GMAC Memory Model

The GMAC library builds an asymmetric distributed shared memory virtual space for systems formed by general purpose CPUs and one or several GPUs. Figure 1.1 outlines this shared virtual memory model, where CPUs and GPUs access a common virtual address space. Specifically, each GPU cannot only access those memory locations that are hosted by its own memory.

A major consequence, and advantage, of the GMAC memory model is the lack of memory copy calls (e.g., `clEnqueueReadBuffer()` /`clEnqueueWrite-Buffer()` in applications source code. By removing explicit data transfers in the application code, GMAC eases the task of programming GPU systems. First, GMAC leverages programmers from the burden of tracking which processor (i.e., CPU or GPU) has modified data structures last time, and coding the necessary calls to access data structures used by both CPUs and GPUs in a coherent way. Second, GMAC also avoids the extra coding necessary to only perform data transfers on those systems where CPUs and GPUs share the same physical memory such as in AMD Fusion APUs.

Under the hood, GMAC dynamically detects memory read and write accesses to data structures shared between CPUs and GPUs and asynchronously

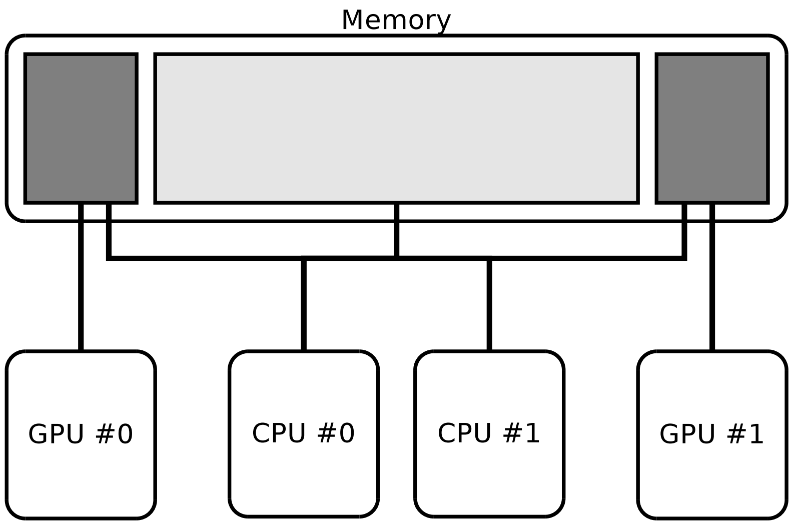


Figure 1.1: Asymmetric Virtual Address Space in GMAC.

updates the contents the GPU while the CPU continues executing the application code. This greedy data transfer mechanism provides performance benefits in discrete GPU systems (i.e., machines where CPUs and GPUs have separated physical memories) because few or none data needs to be transferred from the CPU to the GPU when a kernel is called. Moreover, GMAC tracks data changes using a small granularity, so only those portions of the data that have been effectively modified are transfered. GMAC also avoids copying the data back from the GPU to the CPU after kernel calls until first needed by the CPU, reducing the amount of data transferred. All these asynchronous data copies are automatically triggered and managed by GMAC, without any intervention from the programmer and, thus, effectively simplifying the CPU application code.

GMAC also transparently detects application calls to I/O functions (e.g., `fread()` /`fwrite()`) and provides specialized implementations for these routines that double-buffer data transfers between I/O devices and GPU memory. Analogously, GMAC also double-buffers data transfers between GPUs. This extensive use of double-buffering techniques provides major performance benefits to applications without requiring programmers to implement complex and repetitive code.

1.2 The GMAC CPU Threading Model

GMAC offers two different flavors for integrating GPUs in the application execution. The first model, *HPE*, is a high-level CPU threading model that hides most of the complexities of handling with GPUs in both single-threaded and multi-threaded applications running on top of single-GPU and multi-GPU systems. The second model, *Lite*, is an OpenCL-like interface which fully exposes the OpenCL run-time system to application programmers.

HPE: Heterogeneous Parallel Execution

GMAC/HPE is a high-level model that hides most of the complexity of interacting with GPUs in the CPU code. Figure 1.2 summarizes the HPE model; each CPU thread is bound to a virtual GPU, which is the only GPU each CPU thread can interact with. Specifically, each CPU thread can request the following actions to its virtual GPU:

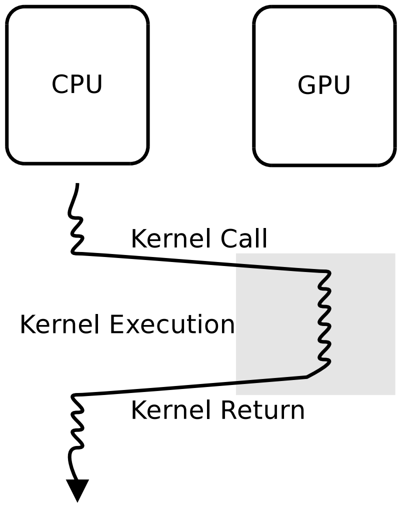


Figure 1.2: HPE CPU Threading model.

- **Memory Allocation /Release.** Each CPU thread can allocate GPU memory which will be accessible by CPU code of any other CPU thread, and by the kernels executed by the CPU thread requesting the memory allocation.
- **Kernel calls.** Each CPU thread can execute kernels in its virtual GPU, which can access to any GPU memory allocated by the calling CPU thread.

All GPU operations in HPE are synchronous function calls; whenever a GPU operation is requested, the CPU code will not continue executing until the GPU operation is done. In HPE, GPU kernel calls and function calls are analogous and interchangeable, and present the following properties:

- The code after a kernel call is guaranteed to not be executed until the kernel has finished executing.
- Any modification within the kernel code to parameters passed by-value has no visibility outside the scope of the kernel.

- Any modification within the kernel code to parameters passed by-reference has visibility outside the scope of the kernel.

Lite

GMAC/Lite is an OpenCL-like interface to GMAC, which implements exactly the same CPU threading model than OpenCL. In Lite, CPU threads create OpenCL contexts and command queues as in any other OpenCL application, but there are new API calls to **allocate and release memory in a OpenCL context**. Any CPU thread can allocate memory which is accessible from any CPU thread and from kernels executed in the OpenCL context where the is allocated. A key difference between this GMAC call and standard OpenCL allocation calls using the `CL_MEM_USE_HOST_PTR` or `CL_MEM_ALLOC_HOST_PTR` is that memory allocated with GMAC does not require being mapped/unmapped to be accessible from the CPU and/or from the GPU. Another major difference between GMAC and OpenCL memory allocation calls is that Lite can be only used from OpenCL contexts that contains a single device.

1.3 Comparison of HPE and Lite

The two API provided by GMAC, HPE and Lite, provide different benefits and limitations. HPE is a high-level interface that hides the complexity of managing OpenCL platforms, devices, contexts, command queues, events, call-backs, and memory buffers. This is in contrast with Lite, which is an OpenCL extension to allow applications to use automatically coherent memory buffers.

We recommend to use HPE in two different scenarios, when porting existing CPU applications to use OpenCL kernels, and when developing new applications that use OpenCL kernels. Application porting is highly simplified when using HPE because the existent source code do not have to be re-engineered to include OpenCL abstractions, such as platforms, devices, contexts, command queues, events, call-backs, and memory buffers. HPE only requires minor source code refactoring:

- Allocate those data structures used by OpenCL kernels using the memory allocation calls provided by GMAC.
- Modify function calls by kernel calls.

Analogously, applications that are implemented from scratch also benefit from HPE provides well-know synchronous function call/shared memory model, most programmers have been successfully using for the last forty years. HPE allows programmers to design their applications without the constraints imposed by OpenCL: decoupled execution (i.e., platforms, devices, and contexts), asynchronous tasks (i.e., command queues, events, and asynchronous invocations), and distributed memory (i.e., memory buffers). Moreover, HPE provides a simple and elegant interface for multi-threaded and/or multi-accelerator applications, which allows programmers to use well-know inter-thread synchronization primitives (i.e., mutexes, semaphores, barriers, and monitors) to orchestrate the concurrent execution of their applications.

The Lite OpenCL extension is designed to bring the performance and portability benefits of GMAC to applications already coded using OpenCL. These applications are easily ported to GMAC by refactoring the existent code to use the allocation calls provided by Lite and removing the code in charge of data transfers. However, Lite can only be used in applications where each OpenCL context is bound to a single OpenCL device.

Chapter 2

Building GMAC/HPE Applications

This chapter discusses how to program applications using GMAC/HPE. First, simple single-threaded applications are considered to introduce the basics of GMAC/HPE programming. Then, this chapter covers how to use GMAC in multi-threaded and/or multi-GPU applications, as well as how to efficiently use multi-threading to overlap several tasks.

2.1 Single-Threaded Applications

We use a simple vector addition example to illustrate the fundamentals of GMAC programming. In this section, our goal is to build an application that reads two variable-sized input floating point vectors from a file, computes a third output floating point vector as an addition of the two input vectors, and stores that vector in a file.

The Kernel code

Listing 2.1 shows the vector addition kernel code. This code might be stored as a global constant char array (i.e., `const char *`) in the application source code, or as a text file. We will assume the latter option, where the code in Listing 2.1 is in a file called `vecadd.cl` in the same directory than the application binary.

A minor comment about the code in Listing 2.1 is the usage of `__global float const * restrict` for the input vectors `a` and `b`. By explicitly spec-

```
__kernel void vecadd(__global float *c,
                     __global float const * restrict a,
                     __global float const * restrict b)
{
    c[get_global_id(0)] = a[get_global_id(0)]
                        + b[get_global_id(0)];
}
```

Listing 2.1: Vector addition kernel code.

```
#include <gmac/opencl.h>

. . .

    eclError_t error_code;
    error_code = eclCompileSourceFile(kernel_file);
    if(error != eclSuccess) error(error_code);
```

Listing 2.2: GMAC/HPE code to make OpenCL code available to the application.

ifying that the input vectors are constant during the kernel execution (i.e., `const`) and no other pointers are used to access them (i.e., `restrict`), the GPU will cache the contents of that vectors, greatly reducing the kernel execution time.

OpenCL Setup and Kernel Loading and Compilation

Setting up the OpenCL run-time tends to be a tedious and repetitive task which requires several lines of code. Analogously, loading the contents of files containing OpenCL kernels is also another annoyance of OpenCL programming. Listing C.1 shows the necessary source code to setup OpenCL, and load and compile the kernel code. GMAC requires no initialization code and provides a simplified path for kernel loading and compilation, as shown in Listing 2.2.

Allocating and Releasing Data Structures

Data structures used by kernels are allocated using the `eclMalloc(void **, size_t)` API call. The first parameter is a pointer to the variable which will hold the CPU pointer used to access the allocated data structure. The second parameter is the size, in bytes, of the memory to be allocated. Finally, `eclMalloc()` returns an error code specifying whether the allocation succeeded (`eclSuccess`) or the condition that prevented the memory allocation. Listing 2.3 shows the source code required to allocate the input and output vectors.

Note that the CPU pointer returned by GMAC can be passed as a parameter to any other function (e.g., `fread()` in Listing 2.3). This code also uses the GMAC API call `eclFree(void *)`, which is used to release the memory allocated by calls to `eclMalloc()`.

Calling Kernels

GMAC/HPE uses a kernel call interface similar to OpenCL. The programmer is required to set the kernel dimensions, pass the kernel parameters, and call the kernel. Listing 2.4 shows the code required to invoke the vector addition kernel.

First, a handler to the kernel to be called is obtained using `__eclKernelGet()`. This handler is used to pass the parameters to the kernel (`__eclKernelSetArg()`, and set the kernel dimensions and call the kernel (`__eclKernelLaunch()`).

Note that data structures used by the kernel need to be first converted to valid OpenCL objects using `eclPtr()`, which takes the CPU memory address returned by `eclMalloc()`.

Final Details

After calling the vector addition kernel, the application writes the output vector to a file, and releases the allocated memory using `eclFree()`, as shown in Listing 2.5.

2.2 Domain Decomposition Across Multiple GPUs

Domain decomposition is a technique that splits a given computation across two or more GPUs. This technique is typically used for two different purposes:

- To speed-up kernel execution by exploiting the computational power of a higher number of GPUs.
- To overcome GPU memory capacity limitations, by distributing input and output data structures across several GPUs.

We illustrate how to implement domain decomposition in GMAC/HPE using the previous vector addition example. In this example we will assume

```c
int load_vector(const char *file_name, float *vector, size_t size)
{
    FILE *fp;
    size_t n;

    fp = fopen(vector_a_file, "b");
    if(fp == NULL) return -1;
    n = fread(a, vector_size, sizeof(float), fp);
    fclose(fp);

    if(n != vector_size) return -1;
    return n;
}

. . .

    float *a, *b, *c;
    ecl_error error_code;

    /* Allocate the input and output vectors */
    error_code = eclMalloc((void **)&a,
                           vector_size * sizeof(float));
    if(error_code != eclSuccess) return error(error_code);
    error_code = eclMalloc((void **)&b,
                           vector_size * sizeof(float));
    if(error_code != eclSuccess) {
        eclFree(a); return error(error_code);
    }
    error_code = eclMalloc((void **)&c,
                           vector_size * sizeof(float));
    if(error_code != eclSuccess) {
        eclFree(a); eclFree(b);
        return error(error_code);
    }

    /* Initialize the input vectors */
    if(load_vector(vector_a_file, a) < 0) {
        fprintf(stderr, "Error␣loading␣%s\n", vector_a_file);
        eclFree(a); eclFree(b); eclFree(c); abort();
    }
    if(load_vector(vector_b_file, b) < 0) {
        fprintf(stderr, "Error␣loading␣%s\n", vector_b_file);
        eclFree(a); eclFree(b); eclFree(c); abort();
    }
```

Listing 2.3: GMAC/HPE code to allocate and initialized the input and output vectors.

```
ecl_kernel kernel;
ecl_error error_code;
cl_mem mem;
cl_uint global_size = vec_size;

error_code = eclGetKernel("vecAdd", &kernel);
if(error_code != eclSuccess) return error(error_code);

mem = cl_mem(eclPtr(c));
error_code = eclSetKernelArg(&kernel, 0, sizeof(mem), &mem);
if(mem == NULL || error_code != eclSuccess)
    return error(error_code);
mem = cl_mem(eclPtr(a));
error_code = eclSetKernelArg(&kernel, 1, sizeof(mem), &mem);
if(mem == NULL || error_code != eclSuccess)
    return error(error_code);
mem = cl_mem(eclPtr(b));
error_code = eclSetKernelArg(&kernel, 2, sizeof(mem), &mem);
if(mem == NULL || error_code != eclSuccess)
    return error(error_code);

error_code = eclCallNDRange(&kernel, 1, NULL,
                            &global_size, NULL);
if(error_code != eclSuccess)  return error(error_code);
```

Listing 2.4: GMAC/HPE code to call the vector addition kernel.

```
FILE *fp;

/* Release input vectors */
eclFree(a);
eclFree(b);

/* Write and release the output vector */
fp = fopen(vector_c_file, "w");
if(fp == NULL) {
    fprintf(stderr, "Cannot␣write␣output␣%s\n",
            vector_c_file);
    eclFree(c); abort();
}
fwrite(c, vector_size, sizeof(float), fp);
fclose(fp);
eclFree(c);
```

Listing 2.5: GMAC/HPE code to write the output vector and release memory.

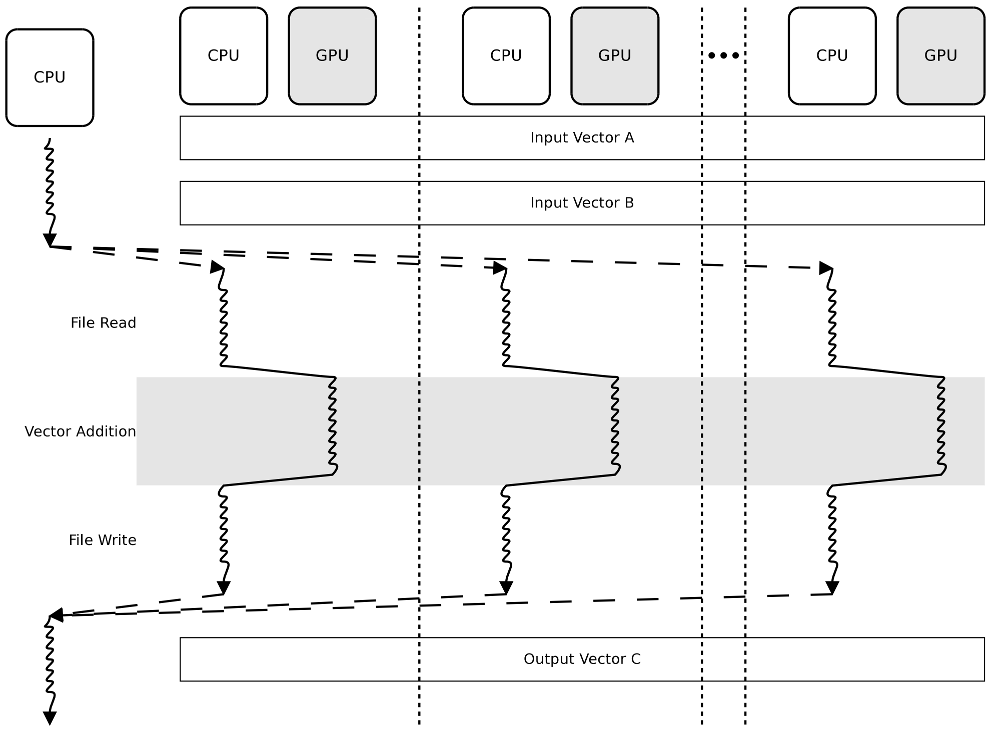


Figure 2.1: Domain decomposition of a vector addition in GMAC/HPE.

that the number of GPUs present in the system is not known until execution-time.

Domain decomposition in GMAC/HPE is implemented using multiple CPU threads. This is in contrast to OpenCL, which allows for a single-threaded implementation. However, good software engineering practices always recommend to use multi-threaded domain decomposition to accomplish modularity, debuggabilty, and maintainability.

We will use the threading scheme in Figure 2.1, where the main CPU thread acts as a control thread that spawns as many worker threads as GPUs are present in the system. Each worker thread is in charge of reading one tile of the input vectors, and compute and write one tile of the output vector. This design allows us to re-use the single-threaded version of vector addition; each worker thread executes that code, with the exception that the file pointer is advanced to the offset corresponding to the tile the thread is computing for each file. Listing 2.6 shows the code for the control thread, which uses `eclGetNumberOfAccelerators()` to get the number of GPUs present in the

```
struct thread_work {
    pthread_t tid;
    size_t offset;
}

. . .

    size_t num_gpus, items_per_thread;
    unsigned n, m;
    int ret;
    struct thread_work *threads;

    num_gpus = eclGetNumberOfAccelerators();
    items_per_thread = vector_size / num_gpus;
    if(vector_size % num_gpus) items_per_thread++;

    threads =
        (struct thread_work *)
        malloc(num_gpus * sizeof(struct thread_work));
    if(threads == NULL) {
        fprintf(stderr, "Not␣enough␣host␣memory\n");
        abort();
    }

    /* Spawn the threads */
    for(n = 0; n < num_gpus; n++) {
        threads[n].offset = n * items_per_thread;
        ret = pthread_create(&threads[n].tid, NULL,
                             vector_add, (void *)&threads[n]);
        if(ret != 0) {
            fprintf(stderr, "Error␣spawing␣threads\n");
            break;
        }
    }
    /* Wait for the threads */
    for(m = 0; m < n; m++) pthread_join(&threads[m].tid, NULL);

    free(threads);
    if(n != m) return -1;

    return 0;
```

Listing 2.6: GMAC/HPE code the control thread when using domain decomposition.

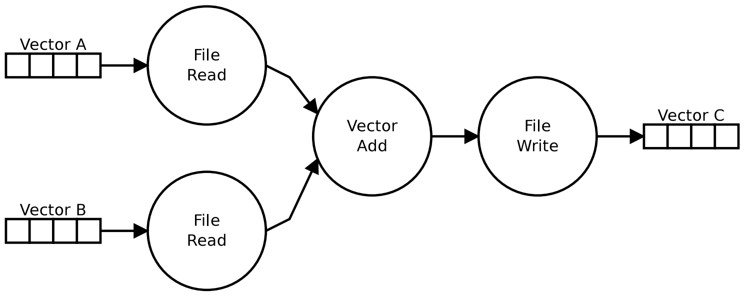


Figure 2.2: Task decomposition for the vector addition example.

system.

Often time, data exchange between domains is necessary. In such as case, common synchronization primitives (e.g., semaphores) become necessary to ensure that the data is not being modified by any thread. Data exchange in GMAC/HPE is implemented through calls to `eclMemcpy()` which takes the common destination, source, and size parameters.

2.3 Task Decomposition

Task decomposition is an implementation technique where applications are divided in units of work (i.e., tasks), which might have inter-dependencies among them (i.e., the output of a task is the input for another task). If none of the inputs of one task depends on the outputs of another task, these two tasks are independent and, therefore, can be executed in parallel. Hence, task decomposition is an implementation technique that provides an amenable way of exploiting parallel execution.

Software pipelining is a very common usage pattern of task decomposition in those applications where one or several tasks can independently process blocks of their input data. A quite popular example are video processing applications, where image frames can be processed one after the other.

Figure 2.2 shows a potential task decomposition for our vector addition example. The application is divided in three tasks: *file read*, *vector addition*, and *file write*. In order to obtain performance improvements, the input vectors are divided in several blocks, being each input data block processed independently. Moreover, the *file read* task is duplicated so each task will read one of the input vectors.

The most common practice to implement task-based applications is to encapsulate each data structure and two semaphores in a single type, as illus-

```
typedef struct {
    float *ptr;
    semaphore_t ready;
    semaphore_t reuse;
} vector_block_t;
```

Listing 2.7: Data type used to encapsulate data structures in the task-based vector addition example.

trated in Listing 2.7 for the task-based implementation of the vector addition example. The `ready` semaphore signals dependent tasks that the data structure is ready to be consumed. Each task executes as many *post* operations over the `ready` semaphore as dependent tasks take the data structure as input. The `reuse` semaphore is only required if data structures are reused during the application execution, which is a common technique to avoid the overheads of memory allocation. Producer tasks perform as many *wait* operations of the `reuse` semaphore as dependent tasks take the data structure as input parameter. Task consuming data structures perform a *wait* operation over the `ready` semaphore before using each data structure, and a *post* operation over the `reuse` semaphore when data structures are not longer used.

The most simple implementation of task-based application consists of a CPU control thread and as many worker CPU threads as task de application is decomposed to. Listing 2.8 shows the initialization code for the CPU control thread in our vector addition example. The CPU control thread allocates the data structures used by the applications, initializing the `ready` and `reuse` semaphores for each of them to zero. Data structures in this application are allocated twice to allow double-buffering of input and output task data structures and, thus, enabling parallel execution of tasks.

Then, the control CPU thread spawns one CPU thread for each task, and then waits for all of them to complete. Listing 2.9 shows the source code for the control thread in the vector addition example, which follows the aforementioned structure. This code is exactly the very same code a task-base application would implement if GMAC/HPE were not used.

Finally, the control code waits for each of the spawned threads to finish, and releases the data structures being used.

The code for worker threads implementing each task, consists of a loop where different blocks of the input and output vectors are processed. This code implements the synchronization mechanism previously discussed using *post* and *wait* operations over the semaphores of the data structures they take as input and output.

```
int allocate_vector_block(vector_block_t *v)
{
    if(sem_init(&v->ready, 0, 0) < 0) return -1;
    if(sem_init(&v->reuse, 0, 0) < 0) goto ready_cleanup;
    if(eclMalloc((void *)&v->ptr, vector_block_size) != eclSuccess)
        goto reuse_cleanup;

    return 0;

reuse_cleanup: sem_destroy(&v->reuse);
ready_cleanup: sem_destroy(&v->ready);
    return -1;
}

. . .

    int ret = -1;

    /* Local data structures */
    vector_block_t vector_a_first, vector_a_second;
    vector_block_t vector_b_first, vector_b_second;
    vector_block_t vector_c_first, vector_c_second;

    /* Allocate first input vector */
    if(allocate_vector_block(&vector_a_first) < 0)
        return -1;
    if(allocate_vector_block(&vector_a_second) < 0)
        goto vector_a_first_cleanup;

    /* Allocate second input vector */
    if(allocate_vector_block(&vector_b_first) < 0)
        goto vector_a_second_cleanup;
    if(allocate_vector_block(&vector_b_second) < 0)
        goto vector_b_first_cleanup;

    /* Allocate output vector */
    if(allocate_vector_block(&vector_c_first) < 0)
        goto vector_b_second_cleanup;
    if(allocate_vector_block(&vector_c_second) < 0)
        goto vector_c_first_cleanup;
```

Listing 2.8: CPU control thread code to initialize data structures for a task-based implementation of vector addition using GMAC/HPE.

```
typedef struct {
    vector_block_t *first, *second;
} vector_load_args_t;

typedef struct {
    vector_block_t *a_first, *a_second;
    vector_block_t *b_first, *b_second;
    vector_block_t *c_first, *c_second;
} vector_addition_args_t;

typedef vector_store_args_t vector_load_args_t;

. . .

    pthread_t vector_load_a_tid, vector_load_b_tid;
    pthread_t vector_addition_tid, vector_store_tid;

    vector_load_args_t vector_load_a_args, vector_load_b_args;
    vector_addition_args_t vector_addition_args;
    vector_store_args_t vector_store_args;

    /* Task to load the first input vector */
    vector_load_a_args.first = &vector_a_first;
    vector_load_a_args.second = &vector_a_second;
    if(pthread_create(&vector_load_a_tid, NULL, vector_load,
        (void *)&vector_load_a_args) != 0) goto vector_c_second_cleanup;

    /* Task to load the second input vector */
    vector_load_b_args.first = &vector_b_first;
    vector_load_b_args.second = &vector_b_second;
    if(pthread_create(&vector_load_b_tid, NULL, vector_load,
        (void *)&vector_load_b_args) != 0) goto wait_vector_load_a;

    /* Task to load the second input vector */
    vector_addition_args.a_first = &vector_a_first;
    vector_addition_args.a_second = &vector_a_second;
    vector_addition_args.b_first = &vector_b_first;
    vector_addition_args.b_second = &vector_b_second;
    vector_addition_args.c_first = &vector_c_first;
    vector_addition_args.c_second = &vector_c_second;
    if(pthread_create(&vector_addition_tid, NULL, vector_addition,
        (void *)&vector_addition_args) != 0) goto wait_vector_load_b;

    /* Task to load the second input vector */
    vector_store_args.first = &vector_c_first;
    vector_store_args.second = &vector_c_second;
    if(pthread_create(&vector_store_tid, NULL, vector_store,
        (void *)&vector_store_args) != 0) goto wait_vector_addition;
```

Listing 2.9: CPU control thread in a task-based implementation of vector addition.

Chapter 3

Building GMAC/CL Applications

This chapter discusses how to use the OpenCL extensions provided by GMAC/CL. The programming of multi-threaded and complex applications in GMAC/CL does only differ from OpenCL in the way data structures are allocated and passed to kernels. Hence, we only discuss these two topics in the context of the single-threaded implementation of the same vector addition example used in Chapter 2.

3.1 Single-Threaded Applications

Initialization

GMAC/CL requires data allocations to be performed over single-device OpenCL contexts. This simple restriction allows the initialization of GMAC/CL applications to be greatly simplified compared to regular OpenCL applications (Listing C.1). GMAC/CL offers a convenient API call (`clInitHelpers()`). This API call can be also combined with `clHelperLoadProgramFromFile()` to further simplify the load and compilation of OpenCL kernels from external files. The combination of these both GMAC/CL API calls can be combined as illustrated in Listing 3.1.

The `clInitHelpers()` takes a reference to a `size_t` variable where the number of available platforms in the system. A call to `clGetHelpers` will return an array of helpers (one per platform in the system). A helper is a structure that contains the identifier of the platform, an array of devices (`devices`),

```
#include <gmac/cl.h>

. . .

    cl_helper helper;
    size_t platforms;
    cl_program program;
    cl_int error_code;

    /* Initialize OpenCL */
    if(oclInitHelpers(&platforms) != CL_SUCCESS) return -1;
    if(platforms == 0) return -1;

    /* Get helper of the first available platform */
    helper = clGetHelpers()[0];

    /* Load and compile the OpenCL kernel for the platform */
    program = clHelperLoadProgramFromFile(helper, kernel_file, &error_code);
    if(error_code != CL_SUCCESS) {
        clReleaseHelpers();
        return -1;
    }
```

Listing 3.1: GMAC/CL code to make OpenCL code available to the application.

the number of devices (`num_devices`), an array of contexts (`contexts`), which are bound to each of the devices in the system, and an array of command queues (`command_queues`), which are bound to each of the contexts. GMAC/CL applications are not required to use the `cl_helper` structure and the `clInitHelpers()` function, but its usage reduces amount of application initialization code.

Memory Allocation

Memory allocation in GMAC/CL is done using the `clMalloc()` API call, which returns a CPU pointer that can be used in any place of the CPU code. Listing 3.1 shows the GMAC/CL code to allocate data structures using `clMalloc()`. In this example we use the same `load_vector()` function used in Listing 2.2.

Calls to `clMalloc()` take as first parameter an OpenCL context. Listing 3.2 uses one of the contexts created by the call to `clInitHelpers()`, but any valid OpenCL context might be used. Besides the OpenCL context, `clMalloc()` also takes a reference to the pointer where to store the address

```
    float *a, *b, *c;
    ocl_error error_code;

    /* Allocate the input and output vectors */
    error_code = clMalloc(helper.context[0], (void **)&a,
                          vector_size * sizeof(float));
    if(error_code != oclSuccess) return error(error_code);
    error_code = clMalloc(helper.context[0], (void **)&b,
                          vector_size * sizeof(float));
    if(error_code != oclSuccess) {
        clFree(helper.context[0], a);
        return error(error_code);
    }
    error_code = clMalloc(helper.context[0], (void **)&c,
                          vector_size * sizeof(float));
    if(error_code != oclSuccess) {
        clFree(helper.context[0], a); clFree(helper.context[0], b);
        return error(error_code);
    }

    /* Initialize the input vectors */
    if(load_vector(vector_a_file, a) < 0) {
        fprintf(stderr, "Error␣loading␣%s\n", vector_a_file);
        clFree(helper.context[0], a);
        clFree(helper.context[0], b);
        clFree(helper.context[0], c); abort();
    }
    if(load_vector(vector_b_file, b) < 0) {
        fprintf(stderr, "Error␣loading␣%s\n", vector_b_file);
        clFree(helper.context[0], a);
        clFree(helper.context[0], b);
        clFree(helper.context[0], c); abort();
    }
```

Listing 3.2: GMAC/CL code to allocate and initialized the input and output vectors.

```
    cl_mem mem;
    cl_kernel kernel;
    cl_uint global_size = vector_size;

    kernel = clCreateKernel(program, "vecAdd", &error_code);
    if(error_code != CL_SUCCESS) return error(error_code);

    mem = clBuffer(context, c);
    error_code = clSetKernelArg(kernel, 0, sizeof(cl_mem), &mem);
    if(mem == NULL || error_code != CL_SUCCESS)
        return error(error_code);
    mem = clBuffer(context, a);
    error_code = clSetKernelArg(kernel, 1, sizeof(cl_mem), &mem);
    if(mem == NULL || error_code != CL_SUCCESS)
        return error(error_code);
    mem = clBuffer(context, b);
    error_code = clSetKernelArg(kernel, 2, sizeof(cl_mem), &mem);
    if(mem == NULL || error_code != CL_SUCCESS)
        return error(error_code);

    error_code = clEnqueueNDRangeKernel(command_queue, kernel, 1,
                                        NULL, &global_size, NULL,
                                        0, NULL, NULL);
    if(error_code != CL_SUCCESS) return error(error_code);
    error_code = clFinish(helper.command_queue[0]);
    if(error_code != CL_SUCCESS) return error(error_code);
```

Listing 3.3: GMAC/CL code to call the vector addition kernel.

of the allocated memory, and the number of bytes of memory to be allocated.

Kernel Calls

Kernel calls in GMAC/CL is very similar to both GMAC/HPE and OpenCL. Listing 3.3 shows the code to perform the kernel call, which uses `clBuffer()` to get the `cl_mem` object associated to memory allocated through calls to `clMalloc()`.

Calls to `clBuffer()` takes as first parameter the context which the memory belongs to, and as a second parameter, a CPU pointer returned by `clMalloc()`. If an invalid pointer is passed as a parameter, `clBuffer()` returns a `cl_mem` object initialized to `NULL` to signal the error condition.

```
    FILE *fp;

    /* Release input vectors */
    clFree(helper.context[0], a);
    clFree(helper.context[0], b);

    /* Write and release the output vector */
    fp = fopen(vector_c_file, "w");
    if(fp == NULL) {
        fprintf(stderr, "Cannot␣write␣output␣%s\n",
                vector_c_file);
        clFree(helper.context[0], c); abort();
    }
    fwrite(c, vector_size, sizeof(float), fp);
    fclose(fp);
    clFree(helper.context[0], c);
    clReleaseHelpers();
```

Listing 3.4: GMAC/CL code to write the output vector and release memory.

Memory Release

Listing 3.4 shows the source code to release resources in GMAC/CL. Memory is released calling `clFree()`, while all data structures associated with OpenCL are released calling to `clReleaseHelpers()`.

Calls to `clFree()` require the OpenCL context where the memory was allocated (`cl.context[0]` in Listing 3.4), and the CPU address returned by `clMalloc()`. If an OpenCL context other than the one where the memory was allocated is passed as a parameter to `clFree()`, an `CL_INVALID_CONTEXT` error is returned.

Appendix A

GMAC API

A.1 GMAC/HPE C types for OpenCL

`ecl_error`

Most GMAC calls return an error code. `eclSuccess` is returned on success.

Value	Description
`eclSuccess`	No error
`eclErrorMemoryAllocation`	Error allocating memory
`eclErrorLaunchFailure`	Error launching the kernel
`eclErrorNotReady`	
`eclErrorNoAccelerator`	
`eclErrorInvalidValue`	Invalid value passed to the function
`eclErrorInvalidAccelerator`	Invalid accelerator specified
`eclErrorInvalidAcceleratorFunction`	Invalid accelerator function
`eclErrorInvalidSize`	
`eclErrorAlreadyBound`	
`eclErrorApiFailureBase`	
`eclErrorFeatureNotSupported`	Feature not supported by the hardware/compilation settings
`eclErrorInsufficientAcceleratorMemory`	Not enough accelerator memory available to perform the operation
`eclErrorUnknown`	Unknown error

`ecl_kernel_id`

Constant string with the name of a function to be executed on an accelerator.

`ecl_memory_hint`

Value	Description
`ECL_GLOBAL_MALLOC_CENTRALIZED`	Prefer centralized (host-mapped) implementation for global malloc
`ECL_GLOBAL_MALLOC_DISTRIBUTED`	Prefer distributed copies implementation for global malloc

`ecl_protection`

Value	Description
`ECL_PROT_NONE`	No access rights for the mapping
`ECL_PROT_READ`	Read-only access rights for the mapping
`ECL_PROT_WRITE`	Write-only access rights for the mapping
`ECL_PROT_READWRITE`	Read/Write access rights for the mapping

A.2 GMAC/HPE C API

`ecl_error eclGetKernel(const char *name, ecl_kernel *kernel)`

Description: Gets a kernel handler for the speficied OpenCL kernel.
Parameters

- `name`: Pointer to a NULL-terminated string that contains the name of the kernel
- `kernel`: Pointer to store the address to the kernel handler

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclSetKernelArg(ecl_kernel kernel, unsigned index, size_t size, const void *addr)`

Description: Adds an argument to be used by the following call to eclCallNDRange() on the specified kernel. NOTE: this function does not work for pointer arguments. See eclSetKernelArgPtr.
Parameters

- `kernel`: Kernel handler
- `index`: Index of the parameter being added in the parameter list
- `size`: Size, in bytes, of the argument
- `addr`: Memory address where the param is stored

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclSetKernelArgPtr(ecl_kernel kernel, unsigned index, const void *ptr)`

Description: Adds a pointer argument (returned by eclMalloc or eclGlobalMalloc) to be used by the following call to eclCallNDRange() on the specified kernel
Parameters

- `kernel`: Kernel handler
- `index`: Index of the parameter being added in the parameter list
- `ptr`: Pointer to be passed to the kernel

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclCallNDRange(ecl_kernel kernel, size_t workDim, size_t *globalWorkOffset, size_t *globalWorkSize, size_t *localWorkSize)`

Description: Launches a kernel execution with the specified device work configuration
Parameters

- `kernel`: Kernel handler of the kernel to be executed on the accelerator
- `workDim`: Number of elements for the work size arrays.
- `globalWorkOffset`: Array of *workDim* unsigned elements that specifies the work offset for the work items.
- `globalWorkSize`: Array of *workDim* unsigned elements that specifies the global number of work items.

- `localWorkSize`: Array of *workDim* unsigned elements that specifies the number of work items per work group.

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclReleaseRelease(ecl_kernel kernel)`

Description: Releases the resources of the given kernel handler.
Parameters

- `kernel`: Handler of the kernel to be released

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclCompileSource(const char *code, const char *flags = NULL)`

Description: Prepares the OpenCL code to be used by the application.
Parameters

- `code` Pointer to the NULL-terminated string that contains the code

- `flags` Compilation flags or NULL

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclCompileSourceFile(const char *path, const char *flags = NULL)`

Description: Prepares the OpenCL code in the specified file to be used by the application.
Parameters

- `path`: Pointer to a NULL-terminated string pointing to the file with the code to be prepared

- `flags`:

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclCompileBinary(const unsigned char *binary, size_t size, const char *flags = NULL)`

Description: Prepares the OpenCL binary to be used by the application.
Parameters

- `binary`: Pointer to the array that contains the binary code
- `size`: Size in bytes of the array that contains the binary code
- `flags`: Pointer to a NULL-terminated string with the compilation flags or NULL

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclCompileBinaryFile(const unsigned char *binary, const char *flags = NULL)`

Description: Prepares the OpenCL binary in the specified file to be used by the application.
Parameters

- `path`: Pointer to a NULL-terminated string pointing to the file with the binary code to be prepared
- `flags`: Pointer to a NULL-terminated string with the compilation flags or NULL

Returns: eclSuccess on success, an error code otherwise

`unsigned eclGetNumberOfAccelerators()`

Description: Get the number of available accelerators in the system **Returns**: Number of accelerators

`unsigned eclGetFreeMemory()`

Description: Gets the amount of available accelerator memory in bytes **Returns**: Amount (in bytes) of the available accelerator memory `ecl_error eclMigrate(unsigned acc)`

Description: Attach the calling CPU thread to the given accelerator **Parameters**

- `acc`: Id of the accelerator to attach to

Returns: eclSuccess on success, an error code otherwise

`ecl_error eclMalloc(void **devPtr, size_t count)`

Description: Allocate shared memory

- `devPtr`: Memory address of the pointer to store the allocated memory
- `count`: Size (in bytes) of the memory to be allocated

Returns: eclSuccess on success, an error code otherwise

`eclGlobalMalloc(void **devPtr, size_t count, ecl_memory_hint hint dv(ECL_GLOBAL_MALLOC_CENTRALIZED))`

Description: Allocate shared memory accessible from all accelerators

- `devPtr`: Memory address of the pointer to store the allocated memory
- `count`: Size (in bytes) of the memory to be allocated
- `hint`: Type of desired global memory

Returns: eclSuccess on success, an error code otherwise

`eclFree(const void *cpuPtr)`

Description: Release shared memory

- `cpuPtr`: Shared memory address to be released

Returns: eclSuccess on success, an error code otherwise

`eclThreadSynchronize()`

Description: Wait until all previous accelerator calls are completed **Returns**: eclSuccess on success, an error code otherwise `eclGetLastError()`

Description: Get the last error produced by GMAC **Returns**: eclSuccess on success, an error code otherwise

`void *eclMemset(void *cpuPtr, int c, size_t count)`

Description: Initialize a shared memory region

- `cpuPtr`: Starting shared memory address.
- `c`: Value used to be initialized
- `count`: Size (in bytes) of the shared memory region to be initialized

Returns: Shared memory address that has been initialized

`void *eclMemcpy(void *cpuDstPtr, const void *cpuSrcPtr, size_t count)`

Description: Copy data between shared memory regions

- `cpuDstPtr`: Destination shared memory
- `cpuSrcPtr`: Source shared memory
- `count`: Size (in bytes) to be copied

Returns: Destination shared memory address

A.3 GMAC/HPE C++ types for OpenCL

All symbols in the C++ API are nested in the `ecl` namespace.

`ecl::error_t`

Most GMAC calls return an error code. `eclSuccess` is returned on success.

Value	Description
`eclSuccess`	No error
`eclErrorMemoryAllocation`	Error allocating memory
`eclErrorLaunchFailure`	Error launching the kernel
`eclErrorNotReady`	
`eclErrorNoAccelerator`	
`eclErrorInvalidValue`	Invalid value passed to the function
`eclErrorInvalidAccelerator`	Invalid accelerator specified
`eclErrorInvalidAcceleratorFunction`	Invalid accelerator function
`eclErrorInvalidSize`	
`eclErrorAlreadyBound`	
`eclErrorApiFailureBase`	
`eclErrorFeatureNotSupported`	Feature not supported by the hardware/compilation settings
`eclErrorInsufficientAcceleratorMemory`	Not enough accelerator memory available to perform the operation
`eclErrorUnknown`	Unknown error

`ecl::kernel_id_t`

Constant string with the name of a function to be executed on an accelerator.

`ecl::memory_hint`

Value	Description
`ECL_GLOBAL_MALLOC_CENTRALIZED`	Prefer centralized (host-mapped) implementation for global malloc
`ECL_GLOBAL_MALLOC_DISTRIBUTED`	Prefer distributed copies implementation for global malloc

`ecl::protection`

Value	Description
`ECL_PROT_NONE`	No access rights for the mapping
`ECL_PROT_READ`	Read-only access rights for the mapping
`ECL_PROT_WRITE`	Write-only access rights for the mapping
`ECL_PROT_READWRITE`	Read/Write access rights for the mapping

A.4 GMAC/HPE C++ API

`ecl::error_t ecl::kernel::kernel()`

Description: Creates a OpenCL kernel handler.

`ecl::kernel::kernel(std::string name, ecl::error &err)`

Description: Creates and gets an OpenCL kernel handler.
Parameters

- `name`: string with the name of the OpenCL kernel to be retrieved
- `err`: Reference to store the error code. eclSuccess on success, an error code otherwise

```
template <typename T>
ecl::error_t ecl::kernel::setArg(unsigned index, size_t
size, const T *addr)
```

Description: Adds an argument to be used by the following call to launch()
Parameters

- `index`: Index of the parameter being added in the parameter list
- `size`: Size, in bytes, of the argument
- `addr`: Memory address where the param is stored

Returns: eclSucces on success, an error code otherwise

```
template <typename T>
ecl::error_t ecl::kernel::setArg(unsigned index, T val)
```

Description: Adds an argument to be used by the following call to call-NDRange()
Parameters

- `index`: Index of the parameter being added to the parameter list
- `val`: Parameter to be stored

Returns: eclSucces on success, an error code otherwise `ecl::error_t callNDRange(size_t workDim, size_t *globalWorkOffset, size_t *globalWorkSize, size_t *localWorkSize)`

Description: Launches a kernel execution with the specified device work configuration
Parameters

- `workDim`: Number of elements for the work size arrays.
- `globalWorkOffset`: Array of *workDim* unsigned elements that specifies the work offset for the work items.
- `globalWorkSize`: Array of *workDim* unsigned elements that specifies the global number of work items.
- `localWorkSize`: Array of *workDim* unsigned elements that specifies the number of work items per work group.

Returns: eclSucces on success, an error code otherwise

`ecl::error ecl::compileSource(const char *code, std::string flags = "")`

Description: Prepares the OpenCL code to be used by the application.
Parameters

- `code` String that contains the code to be prepared
- `flags`: String that contains the compilation flags or empty string

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl::compileSourceFile(std::string path, std::string flags = "")`

Description: Prepares the OpenCL code in the specified file to be used by the application.
Parameters

- `path`: String that contains the path to the file with the source code to be prepared
- `flags`: String that contains the compilation flags or empty string

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl::compileSourceStream(std::istream &in, std::string flags = "")`

Description: Prepares the OpenCL code in the specified stream to be used by the application.
Parameters

- `path`: Reference to an input stream

- `flags`: String that contains the compilation flags or empty string

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl::compileBinary(const unsigned char *binary, size_t size, std::string flags = "")`

Description: Prepares the OpenCL binary to be used by the application.
Parameters

- `binary`: Pointer to the array that contains the binary code

- `size`: Size in bytes of the array that contains the binary code

- `flags`: String that contains the compilation flags or empty string

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl::compileBinaryFile(std::string path, std::string flags = "")`

Description: Prepares the OpenCL code in the specified file to be used by the application.
Parameters

- `path`: String that contains the path to the file with the binary code to be prepared

- `flags`: String that contains the compilation flags or empty string

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl::compileBinaryStream(std::istream &in, std::string flags = "")`

Description: Prepares the OpenCL code in the specified stream to be used by the application.
Parameters

- `in`: Reference to an input stream that contains the binary code
- `flags`: String that contains the compilation flags or empty string

Returns: eclSuccess on success, an error code otherwise

`unsigned ecl::getNumberOfAccelerators()`

Description: Get the number of available accelerators in the system **Returns**: Number of accelerators

`unsigned ecl::getFreeMemory()`

Description: Gets the amount of available accelerator memory in bytes **Returns**: Amount (in bytes) of the available accelerator memory

`ecl::error ecl::migrate(unsigned acc)`

Description: Attach the calling CPU thread to the given accelerator **Parameters**

- `acc`: Id of the accelerator to attach to

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl::malloc(void **devPtr, size_t count)`

Description: Allocate shared memory

- `devPtr`: Memory address of the pointer to store the allocated memory
- `count`: Size (in bytes) of the memory to be allocated

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl::globalMalloc(void **devPtr, size_t count, ecl_memory_hint hint dv(ECL_GLOBAL_MALLOC_CENTRALIZED))`

Description: Allocate shared memory accessible from all accelerators

- `devPtr`: Memory address of the pointer to store the allocated memory
- `count`: Size (in bytes) of the memory to be allocated
- `hint`: Type of desired global memory

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl:free(const void *cpuPtr)`

Description: Release shared memory

- `cpuPtr`: Shared memory address to be released

Returns: eclSuccess on success, an error code otherwise

`ecl::error ecl::threadSynchronize()`

Description: Wait until all previous accelerator calls are completed **Returns**: eclSuccess on success, an error code otherwise

`ecl::error ecl::getLastError()`

Description: Get the last error produced by GMAC **Returns**: eclSuccess on success, an error code otherwise

`void *ecl::memset(void *cpuPtr, int c, size_t count)`

Description: Initialize a shared memory region

- `cpuPtr`: Starting shared memory address.
- `c`: Value used to be initialized
- `count`: Size (in bytes) of the shared memory region to be initialized

Returns: Shared memory address that has been initialized `void *ecl::memcpy(void *cpuDstPtr, const void *cpuSrcPtr, size_t count)`

Description: Copy data between shared memory regions

- `cpuDstPtr`: Destination shared memory
- `cpuSrcPtr`: Source shared memory
- `count`: Size (in bytes) to be copied

Returns: Destination shared memory address

Appendix B

Building and Installing GMAC

B.1 GNU/Linux

Requisites

To compile GMAC, the following software must be installed:

- CMake 2.8 or higher (http://www.cmake.org)
- GCC 4.3 or higher (http://gcc.gnu.org)
- AMD APP SDK 2.5 or higher (http://developer.amd.com/gpu/AMDAPPSDK)

Additionally, to build the GMAC documentation, there are the following extra requisites:

- Doxygen (http://www.doxygen.org)
- LaTeX(http://www.latex-project.org) with the following packages: Memoir (document class), graphicx, tabularx, wrapfig, listings, xspace, hyphenat, and hyperref.
- Rubber (https://launchpad.net/rubber)
- GNOME Dia (http://projects.gnome.org/dia)

Compilation and Installation

First you need to extract GMAC from the tarbal file, by typing:

```
tar -jxf gmac-0.0.3.tar.bz2
```

This will create a directory called `gmac`. Although GMAC can be compiled in the same directory than the source file, we recommend to build it in a separate directory.

```
mkdir -p gmac/build
cd gmac/build
```

GMAC uses CMake to generate platform-specific compilation scripts, so the CMake GUI can be used to configure GMAC. However, GMAC includes a shell script to configure GMAC from the command-line without any need to run any GUI. If you are interested on configure GMAC using the CMake GUI, please refer to the configuration process for Windows. The configuration script for GMAC supports the following options:

- `--enable-opencl`
  Enable the compilation for OpenCL
- `--enable-debug`
  Compile GMAC in debug mode, providing symbol information, GMAC built-in extra checking, and no optimizations.
- `--enable-static`
  Compile GMAC as a static library (by default, GMAC gets compiled as a dynamic library)
- `--enable-tests`
  Compile GMAC tests
- `--enable-doc`
  Compile GMAC documentation
- `--enable-trace-console`
  Dump execution traces to the console
- `--with-opencl-include =` *<path>*
  Sets the path to the OpenCL header files
- `--with-opencl-library =` *<path>*
  Sets the path to the OpenCL libraries

Assuming that AMD APP SDK is installed in `/opt/ati`, configure GMAC by executing

```
../configure --enable-opencl --with-opencl-include=/opt/ati/include
--with-opencl-library=/opt/ati/lib
```

Finally, to compile and install GMAC, you just need to type in the command line:

```
make
make install
```

B.2 Microsoft Windows

Requisites

To compile GMAC, the following software must be installed:

- CMake 2.8 or higher (http://www.cmake.org)
- Visual Studio 2008 SP1
- AMD APP SDK 2.5 or higher (http://developer.amd.com/gpu/AMDAPPSDK)

Additionally, to build the GMAC documentation, there are the following extra requisites:

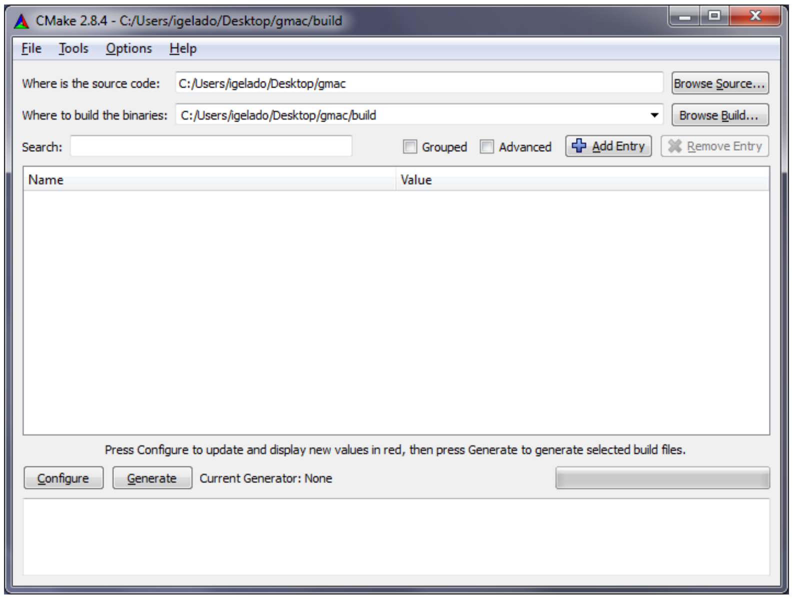
- Doxygen (http://www.doxygen.org)
- Miktex (http://miktex.org) with the following packages: Memoir (document class), graphicx, tabularx, wrapfig, listings, xspace, hyphenat, and hyperref.
- GNOME Dia (http://projects.gnome.org/dia)

Compilation and Installation

First you need to extract GMAC from the zip file, which will create a directory called `gmac`.

GMAC uses CMake to generate platform-specific compilation scripts, so the CMake GUI is used to configure GMAC. Launch CMake-GUI and set the source directory to the path where you extracted GMAC, and the build directory to a build subdirectory inside GMAC, as in Figure B.1, and click the *Configure* button.

A new CMake window (Figure B.2) appears to ask for the compiler to be used; select *Visual Studio 9 2008* to compile GMAC for 32-bits, or *Visual Studio 9 2008 Win64* to compile GMAC for 64-bits.



Where is the source code: C:/Users/igelado/Desktop/gmac

Where to build the binaries: C:/Users/igelado/Desktop/gmac/build

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: None

Figure B.1: Step 1, set the source and build path in CMake-GUI

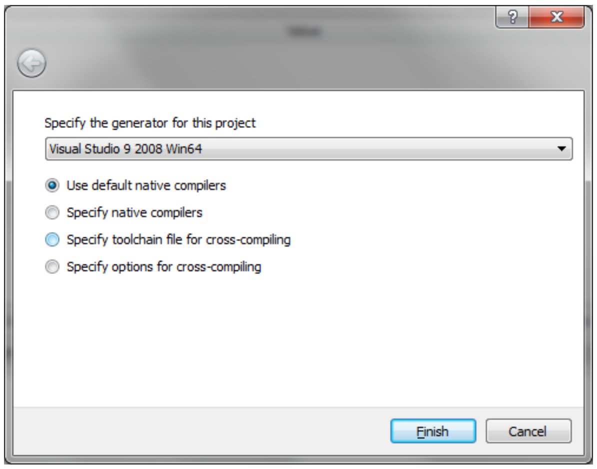
After an initial configuration attempt, CMake will report an error (Figure B.3) because it fails to guess the configuration parameters for the system.

Select `USE_OPENCL`, and `USE_LITE`, as in Figure B.4, and click on *Configure*.

A new error window will appear (Figure B.5) if CMake fails to find the location of the OpenCL header files and libraries.

In such a case, you need to set those paths by hand, as in Figure B.6, a click on *Configure* again. Finally, a new Visual Studio project will be produced inside the `build` folder.

Open the *gmac* Visual Studio project, located in the *build* folder inside GMAC. Inside Visual Studio, set the active configuration to *Release*, and press F7 to compile GMAC. Finally, build the `INSTALL` project to install GMAC in your system.

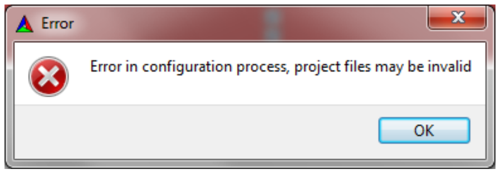


Specify the generator for this project

Visual Studio 9 2008 Win64

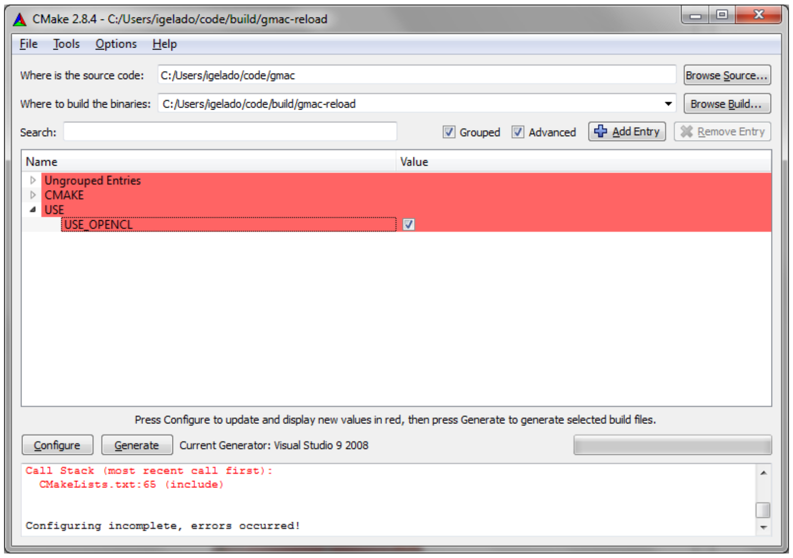
[✓ Use default native compilers]
Specify native compilers
Specify toolchain file for cross-compiling
Specify options for cross-compiling

Figure B.2: Step 2, select the compiler to be used



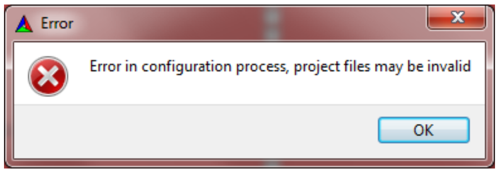
Error in configuration process, project files may be invalid

Figure B.3: Configuration error; CMake is unable to guess the configuration


CMake 2.8.4 - C:/Users/igelado/code/build/gmac-reload
File Tools Options Help
Where is the source code: C:/Users/igelado/code/gmac
Browse Source...
Where to build the binaries: C:/Users/igelado/code/build/gmac-reload
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Name
Value
Ungrouped Entries
CMAKE
USE
USE_OPENCL
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate
Current Generator: Visual Studio 9 2008

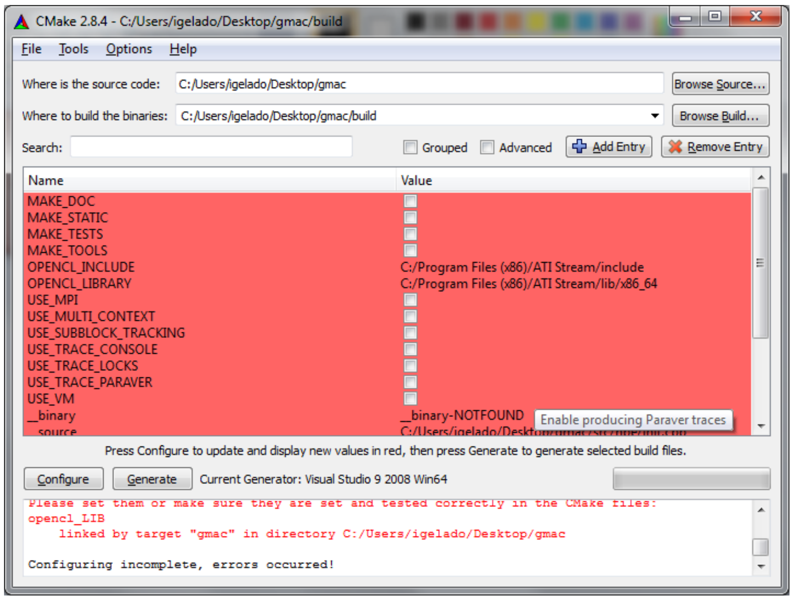
```
Call Stack (most recent call first):
  CMakeLists.txt:65 (include)


Configuring incomplete, errors occurred!
```

Figure B.4: Step 3, set the correct configuration parameters in CMake-GUI



Error in configuration process, project files may be invalid

Figure B.5: Configuration error; CMake does not find OpenCL files



Where is the source code: C:/Users/igelado/Desktop/gmac

Where to build the binaries: C:/Users/igelado/Desktop/gmac/build

Search: ☐ Grouped ☐ Advanced

Name	Value
MAKE_DOC	☐
MAKE_STATIC	☐
MAKE_TESTS	☐
MAKE_TOOLS	☐
OPENCL_INCLUDE	C:/Program Files (x86)/ATI Stream/include
OPENCL_LIBRARY	C:/Program Files (x86)/ATI Stream/lib/x86_64
USE_MPI	☐
USE_MULTI_CONTEXT	☐
USE_SUBBLOCK_TRACKING	☐
USE_TRACE_CONSOLE	☐
USE_TRACE_LOCKS	☐
USE_TRACE_PARAVER	☐
USE_VM	☐
__binary	__binary-NOTFOUND
__source	C:/Users/igelado/Deskt

Enable producing Paraver traces

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 9 2008 Win64

```
Please set them or make sure they are set and tested correctly in the CMake files:
opencl_LIB
    linked by target "gmac" in directory C:/Users/igelado/Desktop/gmac

Configuring incomplete, errors occurred!
```

Figure B.6: Step 4, set the path for the OpenCL header files and libraries

Appendix C

OpenCL Code Listings

C.1 OpenCL Initialization

```
    cl_context context;
    cl_command_queue command_queue;
    cl_program program;
    FILE *fp;
    stat file_stat;
    size_t n;
    char *source_code = NULL;

    /* Create an OpenCL context containing a GPU */
    context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU,
                                      NULL, NULL, &error_code);
    if(error_code != CL_SUCCESS) return error(error_code);

   /* Open the file containing the kernel
     * We need to first get the source file size to allocate
     * the necessary memory
     */
    if(stat(kernel_file, &stat) < 0) {
        clReleaseContext(context);
        return error(CL_INVALID_PROGRAM);
    }
    source_code = (char *)malloc(stat.st_size);
    if(source_code == NULL) {
        clReleaseContext(context);
        return error(CL_OUT_OF_HOST_MEMORY);
    }
    fp = fopen(kernel_file, "rt");
    if(fp == NULL) {
        free(source_code);
        clReleaseContext(context);
        return error(CL_INVALID_PROGRAM);
    }
    n = fread(source_code, stat.st_size, sizeof(char), fp);
    fclose(fp);
    if(n != stat.st_size) {
        free(source_code);
        clReleaseContext(context);
        return error(CL_INVALID_PROGRAM);
    }
    program = clCreateProgramWithSource(context, 1, &source_code,
                                        &n, &error_code);
    free(source_code);
    if(error_code != CL_SUCCESS) {
        clReleaseContext(context);
        return error(error_code);
    }

    /* We get the OpenCL device bound to the context, becase we
     * need that * device to build the program and create the
     * command queue
     */
    error_code = clGetContextInfo(context, CL_CONTEXT_NUM_DEVICES,
                                  sizeof(num_devices), &num_devices,
                                   NULL);
    if(error_code != CL_SUCCESS || num_devices == 0) {
        clReleaseProgram(program); clReleaseContext(context);
        return error(CL_DEVICE_NOT_AVAILABLE);
    }
    devices = (cl_device_id *)
              malloc(num_devices * sizeof(cl_device_id));
    if(devices == NULL) {
        clReleaseProgram(program); clReleaseContext(context);
        return error(CL_OUT_OF_HOST_MEMORY);
    }
    error_code = clGetContextInfo(context, CL_CONTEXT_DEVICES,
                                  num_devices * sizeof(cl_device_id),
                                  devices, NULL);
    if(error_code != CL_SUCCESS) {
        free(devices);
        clReleaseProgram(program); clReleaseContext(context);
        return error(error_code);
    }

    /* Buidl the program */
    error_code = clBuildProgram(program, 1, *devices, NULL,
                                NULL, NULL);
    if(error_code != CL_SUCCESS) {
        free(devices);
        clReleaseProgram(program); clReleaseContext(context);
        return error(error_code);
    }

    /* Finally, we create a command queue to be able to
     * call kernelss
     */
    command_queue = clCreateCommandQueue(context, *devices,
                                         NULL, &error_code);
    free(devices);
    if(error_code != CL_SUCCESS) {
        clReleaseProgram(program); clReleaseContext(context);
        return error(error_code);
    }
```

Listing C.1: OpenCL initialization code, which is not longer required in GMAC.