Universidade Federal de Minas Gerais

ParallelME Runtime: Technical Report

Technical description behind ParallelME's low-level runtime system.

Final Report submitted to LGE as part of the Project *Scheduling heterogeneous tasks on hybrid device environments.*

Belo Horizonte, MG
June 04, 2016

ParallelME Runtime: Technical Report

1 Introduction

This document contains an overview of the ParallelME framework's low-level runtime system. The execution mechanism behind the runtime, details of the technology used and relevant features will be adressed in this document.

2 ParallelME Run-time Overview

The runtime component of ParallelME is responsible for coordinating in an efficient way all available processing units in a mobile architecture. It organizes and manages low level tasks generated by the compiler component, from definitions expressed by developers in the programming abstraction. The implementation is in C++ and was inserted in the Android system using the NDK toolkit.

In general, using the runtime involves the following steps:

1. Identify the computing resources available on the mobile architecture
2. Specify and compile the kernels to be executed
3. Organize the kernels into a task and specify the proper callbacks
4. Submit the task for execution
5. Decide where to execute by following a scheduling policy
6. Configure the kernels to be executed
7. Execute the task on a chosen processing unit
8. Copy the task's output to a host variable

The Runtime system uses OpenCL as the basis to implement all the steps mentioned above. OpenCL is a platform that provides control to every step of parallel programming in heterogeneous architectures, from the identification of the processing units to the task execution on a specific processor. This platform fits perfectly into the architecture of the runtime, as it gives a lot of flexibility to decide how tasks will be executed.

2.1 Runtime Details and Internals

The first Runtime step begins with the identification of available devices in the mobile architecture. On OpenCL, this is done by calling the routine ***clGetPlatformIDs*** to get the available platforms (such as ARM, Qualcomm and Nvidia) and then calling ***clGetDeviceIDs*** for each platform to get the available devices (such as Arm's Mali or Qualcomm's Krait). Finally, an individual context and command queue are created for each device.

As soon as the devices are identified, one *Worker* is created for each of the available devices. This Worker is a class that instantiates a thread that will be responsible for managing the retrieval and execution of work sets for its processing unit. This thread is known as a **Worker thread**. Then, a scheduler class (responsible for a given scheduling policy) is instantiated. Figure 1 shows the dynamics of this first step performed at the Runtime startup.

The first Runtime configuration step is done in the constructor of the `parallelme::Runtime` class. When using the runtime, the only thing that is needed is to create a shared pointer to the runtime, as shown in the code below. Using a shared pointer is important because other classes will need a reference to this class later on. The runtime needs a pointer to the JavaVM structure when being constructed, because it has to link the worker threads to the JavaVM.

```
1 JavaVM *jvm;
2 // <initialize the jvm structure>
3 auto runtime = std::make_shared<parallelme::Runtime>(jvm);
```

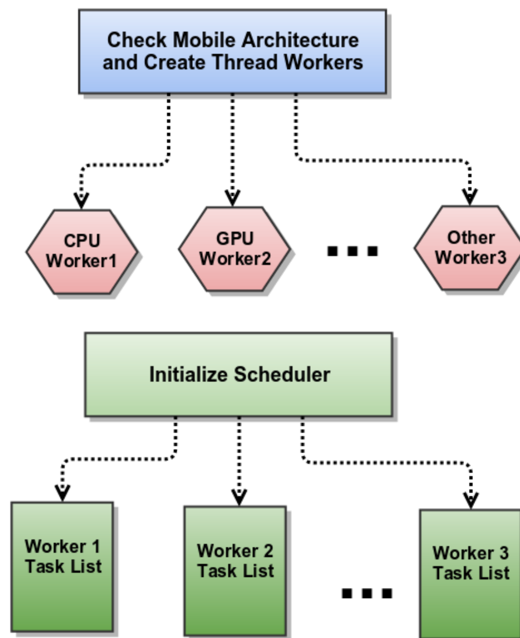

Figure 1: Runtime setup: the runtime identifies the available processing units and the scheduler creates work queues for the processing units, according to its policy.

When created, the scheduler, depending on its policy, creates different queues, one for each processor type (CPU, GPU, accelerator, etc), and a task that is sent to the scheduler for execution has the information of which processing units it can execute at. The runtime can be created with different schedulers by specifying a second paremeter to the Runtime constructor:

```
// FCFS is the default scheduler.
auto runtime = std::make_shared<parallelme::Runtime>(jvm);

// HEFT scheduler.
auto runtime = std::make_shared<parallelme::Runtime>(jvm, std::make_shared<parallelme::SchedulerHEFT>());

// PAMS scheduler.
auto runtime = std::make_shared<parallelme::Runtime>(jvm, std::make_shared<parallelme::SchedulerPAMS>());
```

After the Runtime setup, the next step involves compiling the OpenCL source code that contains the kernels used by the application. This involves creating a *Program* class with the OpenCL kernel source code. The Program class compiles the source code and throws an exception in case of failure with the error message.

```
    auto program = std::make_shared<parallelme::Program>(runtime, source);
```

The Program class is initialized by supplying the OpenCL kernel source code and a pointer to the Runtime. The source code must be a C string.

Note that the Runtime creation and the Program instantiation are done before execution, preferably on the initialization of the application, because finding the devices and compiling the sources can take some time.

On the next step, the kernels are organized in a particular order of execution, using a class called *Task*. It is important to emphasize that these tasks are generic and are not linked, in principle, to any processing unit. When submitted to execution, these tasks will be sent to a scheduler and only after its decision they will be executed on a specific device. In the case of tasks that are composed of a set of kernels, all these kernels will be executed on the same device. The code below shows the creation of a task and how to link kernels to be executed in order:

```
    auto task1 = std::make_unique<parallelme::Task>(program);

    task1->addKernel("to_float")
        ->addKernel("to_yxy")
        ->addKernel("line_log_average")
        ->addKernel("log_average")
        ->addKernel("tonemap")
        ->addKernel("to_rgb")
        ->addKernel("to_bitmap");
```

As mentioned, a task may contain one or more kernels. If more than one kernel is assigned to a task (as exemplified in *_task1*), they are executed in the order they were instantiated: when executing the *_task1*, first the worker will execute the *"to_float"* kernel, then the *"to_yxy"* kernel, the *"line_log_average "* and so on, as shown in figure 2. This mechanism ensures the execution of kernels in order if there is a data dependence between them. Figure 2 also exemplifies that although kernels in the same task are executed sequentially, kernels in different tasks are executed in parallel if the computing resources are available (if there is more than one device and they are not busy executing other tasks).

When creating a task, it is possible to specify how well a task executes on a given device. In the ParallelME Runtime, the lower the score the better is the execution on a device. So, if a given task is good to execute on the GPU, but bad on the CPU, the Task could be created like this:

```
1    auto task = std::make_unique<parallelme::Task>(program, parallelme::Task::Score(2.0f, 0.5f));
```

The first parameter of the Score constructor is the score of the task at the CPU, while the second parameter is the score of the task at the GPU.

Also note that the task must be an unique pointer: this way, once the task is submitted to the Runtime for execution, the user loses all access to the class, as the Runtime will acquire ownership of the pointer.

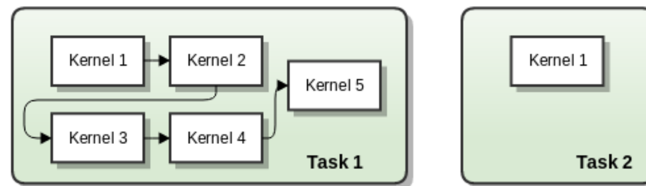

Figure 2: Runtime Task Organization.

To configure the Task before executing it, a callback can be set to prepare all kernels. The runtime uses a callback to configure the kernel because it is a simple and efficient way to store everything that needs to be configured before execution on a specific device. When the task's configuration callback is set, it is not called right away, but stored to be called when the runtime knows where the task will wexecute. This is done after the task is pushed to the scheduler and a worker pops it. As the worker knows which device will be used to execute the kernels, it can call the configuration callback. The configuration callback has the following form:

```
1  task1->setConfigFunction([=] (DevicePtr &device, KernelHash &kernelHash) {
2      // Configuration code...
3  });
```

The configuration callback has access to all the kernels added before to the task, being able to set all the parameters of the kernel call and set the work size of the kernel. The work size is the number of work items that are being processed in the kernel (i.e. the number of pixels in an image if each kernel call processes a single pixel). The code below shows how to set the work size:

```
1  task1->setConfigFunction([=] (DevicePtr &device, KernelHash &kernelHash) {
2      kernelHash["to_float"]->setWorkSize(size);
3  });
```

The kernel function arguments can be set using the index of the parameter in the kernel prototype. If the data is a simple scalar variable, it can be set directly by calling `setArg` with the variable's contents, like the call below, that set's the `to_float` kernel's first parameter with the height scalar variable (notice how the `setArg` and `setWorkSize` calls of a same kernel can be chained together):

```
int height = 42; // or other value
task1->setConfigFunction([=] (DevicePtr &device, KernelHash &kernelHash) {
    kernelHash["to_float"]
        ->setArg(0, height)
        ->setWorkSize(size);
});
```

If the input to the kernel's parameter is not a scalar type or is a pointer, it can not be set directly like in the example above. Instead, a `Buffer` class must be created to hold the input's contents. This is needed because non-scalar types need to be allocated on the device's memory, and function calls are needed to send and receive the data that is stored at this memory region. A Buffer can be constructed just by specifying the size (in bytes) of the memory region:

```
auto buffer = std::make_shared<parallelme::Buffer>(size);
```

To mark data to be copied into the buffer, a `set*Source` function can be used. But |beware|: these functions only save a reference to the data when being called, and the data is only copied when the data is requested, either by a kernel or by a `copyTo*` function. Because of this, the user is responsible for maintaining the data's source available until the kernel starts execution. An example of marking a jarray to be copied is shown below:

```
// The JNIEnv of the current thread and the jarray are needed.
buffer.setJArraySource(env, array);
```

An important remark is that the JNIEnv inside a task's configure function probably will be different to a JNIEnv outside the configure function. To access the JNIEnv of the configure function, `device->JNIEnv()` must be called:

```
auto buffer = std::make_shared<parallelme::Buffer>(size);

// One option is to mark the data to be copied before the config function:
buffer->setJArraySource(env, array);

task1->setConfigFunction([=] (DevicePtr &device, KernelHash &kernelHash) {
    // The other option is to use device->JNIEnv() from inside the config function:
    buffer->setJArraySource(device->JNIEnv(), array);

    // ...
});
```

A buffer can be shared between different tasks, and it will be automatically copied from one device to another in case each task executes in different devices. However, the runtime will make no attempts to ensure order between the tasks: the user is responsible for mantaining order of execution (a `runtime->finish()` function exists to block until all tasks that were submitted to the runtime finish executing). Also, the user must guarantee that the buffer is not used in two tasks concurrently, or data races and crashes may occur.

To make a buffer the input of a kernel parameter, it just needs to be specified in the `setArg` function:

```
auto buffer = std::make_shared<parallelme::Buffer>(size);
buffer->setJArraySource(env, array);
int height = 42;

task1->setConfigFunction([=] (DevicePtr &device, KernelHash &kernelHash) {
    kernelHash["to_float"]
        ->setArg(0, height)
        ->setArg(1, buffer) // Setting the second kernel parameter to point to the buffer.
        ->setWorkSize(size);
});
```

A second callback can be set to gather the execution results: the finish callback. It is called after all the kernels of a given task finish execution, and can be used to cleanup and copy the output data:

```
task1->setFinishFunction([=] (DevicePtr &device, KernelHash &kernelHash) {
    buffer->copyToJArray(device->JNIEnv(), output); // Copies the buffer content to the output java array.
});
```

These two callbacks should be set before dispatching the task to the scheduler. The task can be submitted to the runtime by the `submitTask()` function. Note that the task must be moved into the runtime. Because of this, the user loses any access to the task:

```
runtime->submitTask(std::move(task1));
```

As an alternative to using the finish callback, the user can call `runtime->finish()` and access the buffer directly after that, as it will have been modified by the task's execution:

```
auto buffer = std::make_shared<parallelme::Buffer>(size);
task->setConfigFunction([=] (DevicePtr &device, KernelHash &kernelHash) {
    kernelHash["fill_with_zeros"]
        ->setArg(0, buffer)
        ->setWorkSize(size);
});

runtime->submitTask(std::move(task1));

// Wait for the runtime to finish executing everything and then copy the data.
runtime->finish();
buffer->copyToJArray(env, outputArray);
```

After submitting a task, other tasks can be submitted before calling the finish() function, so that they can execute in parallel instead of one task at a time.

For a more detailed example of using the runtime please read the Runtime User Manual.

2.2 Runtime Execution Mechanism

In the previous subsection were detailed steps for the Runtime configuration, task creation, parameter linking and task submission. These procedures must be called by the user to implement their application with the Runtime.

After the task submission, the Runtime engine is responsible for managing the task scheduling and execution. Once a task is submitted the **Scheduler API** routine ***pushTask***, following a

particular scheduling policy, allocates the task into a task queue created on the startup scheduler step.

As mentioned in the previous subsection, the Runtime implements **Worker Threads**, responsible for retrieving tasks for their respective devices. When there aren't tasks to be executed, the Worker Threads remain "sleeping". However, when a task is submitted, a signal awakens those threads, which will call the **Scheduler API** routine ***popTask***. This routine is responsible for retrieving a task from the task list, following a particular scheduling policy. When the **Worker Thread** receives the task returned by ***Pop task*** routine it starts the execution process.

The execution begins by calling the routine ***Configure Execution***, set by the user and responsible for linking buffers and scalars to kernel parameters in a task on the specific device they will execute on. After this configuration, the kernel is sent to the underlying programming platform to start the execution on the device. The Worker thread waits for the execution to finish and, at the end, calls the routine ***Finish Execution***, if set by the user, responsible for retrieving the output asynchronously. The Worker thread then asks for the task scheduler if there are any more tasks to execute. If there are, the loop repeats. If not, the thread goes back to sleep.

Figure 3 shows the dynamics of the Runtime, from the Runtime setup and task creation to the scheduling routines and task execution process.

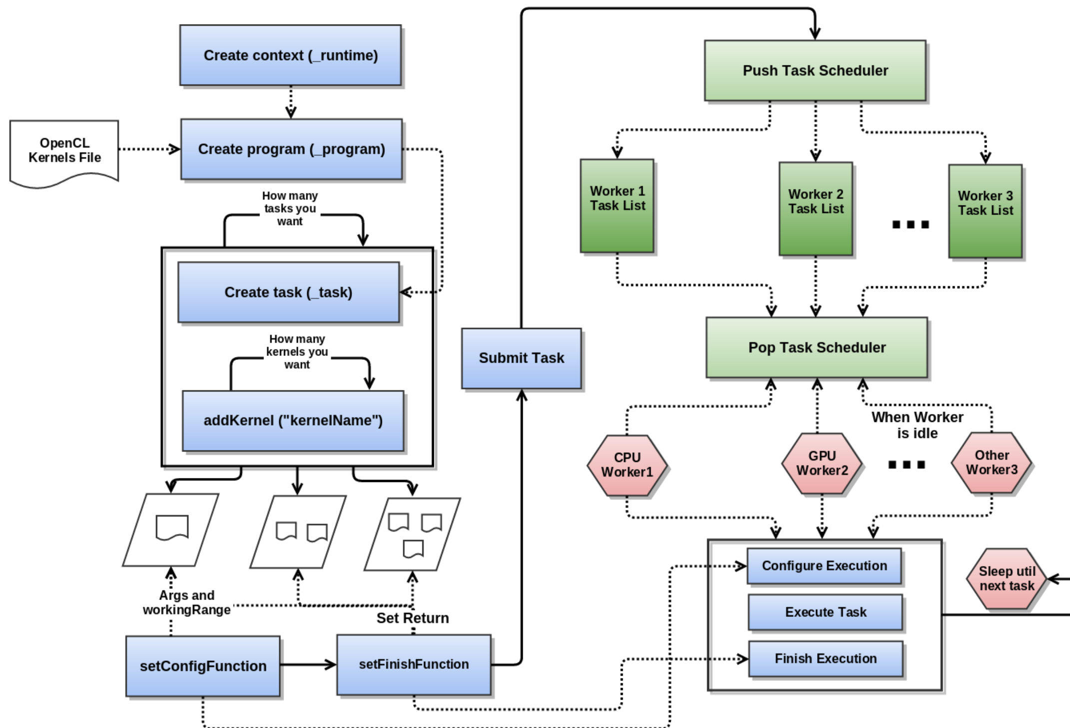

Figure 3: Runtime Execution Mechanism Dynamic

2.3 Runtime Scheduler API

The runtime also provides an API to implement different task schedulers policies. This API involves five main functions that need to be implemented by a new scheduler class. This interface can be found at the `include/parallelme/Scheduler.hpp` file, and a class should inherit this interface and implement it for a new scheduling policy. The five functions are described below:

- **Constructor:** Routine responsible for initializing the structures that will be used by scheduling policy. Typically, this routine is used to create lists and synchronization structures such as semaphores and mutexes.
- **Push Task:** Routine responsible for inserting tasks, following a certain scheduling policy in the queues created during the initialization step. Basically this routine is called when a task has been submitted to the system to be executed.
- **Pop Task:** Routine responsible for get a task at one of the task queues of the scheduling strategy. Basically, when a processing resource is idle, it then calls the Pop Task routine to receive a task to execute.
- **Wait until idle:** Routine called to block and wait until all the tasks pushed to the scheduler are executed. This should call a `wait()` function until the scheduler has no more tasks to pop, when it will call `wakeUp()`.
- **Destructor:** Routine designed to perform some operations when the scheduler class is being destroyed. Basically, in this routine memory areas allocated at startup are released.

Using this interface, a new scheduler can be written and the runtime can be extended to use it by specifying the new scheduler on its constructor. For example, if a hypothetical `MyScheduler` was created (and if it implemented the abstract Scheduler class), it could be used with the runtime in the following way:

```
1 auto runtime = std::make_shared<parallelme::Runtime>(jvm, std::make_shared<MyScheduler>());
```