UDC 004.42
Original Scientific Paper

Utilizing GPGPU in Computer Emulation

Peter Jakubčo *peter.jakubco@tuke.sk*
Technical University of Košice
Faculty of Electrical Engineering and Informatics

Slavomír Šimoňák *slavomir.simonak@tuke.sk*
Technical University of Košice
Faculty of Electrical Engineering and Informatics

Abstract

The article deals with the idea of computer emulation using the GPGPU technology in order to get performance improvements. Basic assumptions for using stream processing in computer emulation effectively are discussed and the structure of an emulator, together with the emulation technique are proposed. The emulator structure, in this case, is of distributed nature, so the communication issues are treated too. As an example demonstrating the viability of the proposal, Random Access Machine (RAM) emulator is given within the paper.

Keywords: Emulation, GPGPU, OpenCL, RAM, stream processing

1. Introduction

There is a variety of reasons for the computer imitation (covering both simulation and emulation), mostly connected with: cross-platform support, backward compatibility preservation, maintaining investments in older software, hardware/software development, experimentation or testing, education purposes or convenience.

More widely known in research community seems to be the term simulation, so we first point at the difference between simulation and emulation. In extreme abstraction, we can state that simulator imitates the microarchitecture of a computer, and if the imitation is accurate enough, the behavior will be preserved as a consequence. An emulator, instead, imitates the behavior primarily and the microarchitecture may be imitated as adding more details into emulation is required [21].

Emulators are more appropriate when the behavior is important as a whole, from the input/output point of view, where it does not matter what internal organization and structure the system has. Within the computer emulation, emulators implement the Instruction Set Architecture (ISA) of a computer [17]. The ISA represents a bridge between the microarchitecture and the programmer. So it is that part of computing machine, which is visible for programmer. The aim at this level of design, is to clarify questions connected with instructions, addressing modes, registers, flags, memory architecture, input/output, etc.

When the emulation of more complex systems (e.g. multi processor systems) is considered, the single processor host computer suffers from significant performance slowdown. If parallel emulation algorithms were used, special parallel computer architectures would be able to perform the emulation without significant performance loss. However, as we want to keep the emulation widely available, our intention is to use standard PCs as the host systems. Our idea therefore is to utilize parallel systems that are increasingly often available within PCs - the current GPUs (Graphics Processing Units).

2. Stream Processing and Computer Emulation

Stream processing is a powerful and relatively new programming model. It does not fit exactly into any well-defined class within current taxonomies. It can be considered as a branch of SIMD (Flynn's taxonomy [3]) or SIMD/MIMD (Duncan's taxonomy [2]) architectures. It is often referred to as SIMT (Single Instruction Multiple Threads), which according to [19] is described as a near MIMD programming model with SIMD implementation efficiency.

In the present time, two kinds of stream processors exist: general-purpose (and stand-alone) processors and specialized stream processors (like GPUs). Most notable general-purpose stream processor is Imagine [10], a flexible architecture developed at Stanford University, funded by Intel and Texas Instruments.

Within the model, applications are expressed as a sequence of computation kernels that operate on streams of data. A kernel is a small, user-developed program that is executed repeatedly on a stream of data. It is a parallel function that operates on each element of input streams to produce output streams which can be fed to subsequent kernels. Each data stream is a variable length collection of records, where each record is a logical cluster of data (Figure 1).

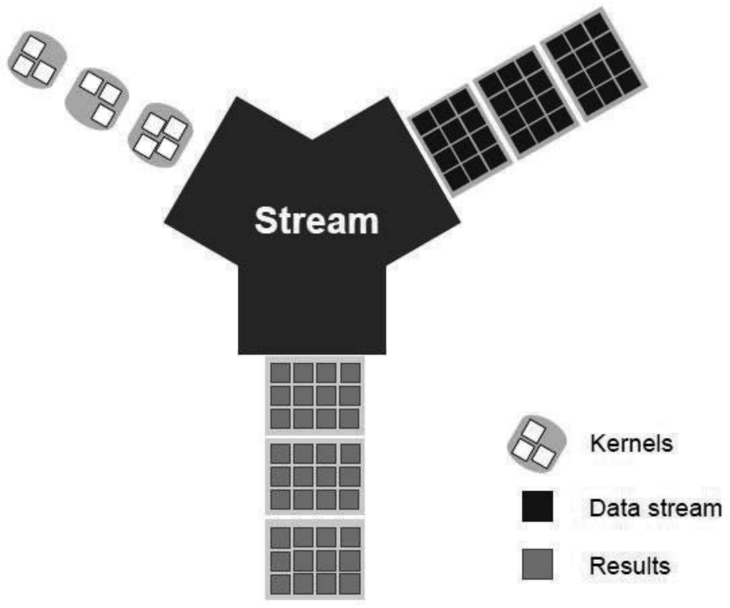


Figure 1: Stream processing as a programming model

According to [18], [23], stream processor exploits three levels of parallelism:

1. Instruction-level - parallelism in the scalar operations within a kernel; stream processors represent VLIW architectures;
2. Data-level - operate on several data items within a stream in parallel;
3. Task level - actually quite limited version of parallelism. Usually same kernels are executed on many executions units, while each execution unit has its own "program counter" or "instruction pointer" register that enables different paths of the same kernel that can be executed on different execution units.

2.1 GPUs as Stream Processors

Recent developments in GPUs include support for programmable shaders which can manipulate vertices and textures in very advanced manner. Because most of these manipulations involve matrix and vector operations, the idea of using GPUs for non-graphical calculations is increasingly often studied nowadays (General-Purpose computing on Graphics Processing Units - GPGPU) [6]. A basic concept here is to use a general purpose graphics processing unit as a modified form of stream processor. This concept turns the massive floating-point computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power, as opposed to being hardwired solely to perform graphical operations.

Most of programming languages for stream processors start with C or C++ and add extensions which provide specific instructions to allow application developers to access kernels and/or streams. Not many streaming languages for GPUs are available today. NVIDIA's CUDA platform [15], [16] is the most widely adopted programming model for GPU computing. The OpenCL [14], [1] is another viable alternative, offered as an open standard, managed by Khronos Group. Next option is the DirectCompute [13] from Microsoft. Short description of CUDA and OpenCL follows.

2.1.1 CUDA

According to [15], CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written in CUDA language. It is a C-like language for developing kernels. Host program written in another language (C, C++, FORTRAN, etc.) calls these kernels. A kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs and outputs.

A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing and result sharing in parallel algorithms. Grids of thread blocks share results in global memory space after kernel-wide global synchronization.

2.1.2 OpenCL

The OpenCL [14] is open and free standard supporting parallel programming of heterogeneous CPUs, GPUs and other processors. It can be considered as a rival of NVIDIA's CUDA or Microsoft's DirectCompute technology. Currently, the OpenCL within ATI SDK [1] can utilize all CPUs that support at least SSE3 extension, and many of ATI or NVIDIA GPUs. Therefore, it can be said that OpenCL is supported more than CUDA. The environments that allow programming of the OpenCL include ATI Stream SDK, NVIDIA OpenCL SDK for the CUDA architecture [16], or OpenCLLink for Mathematica [22]. The architecture of OpenCL fits into the stream processing class and allows both SPMD (Single Program Multiple Data) and MPMD (Multiple Program Multiple Data) programming models (or mix of the two). Organization of an OpenCL device is depicted in the Figure 2.

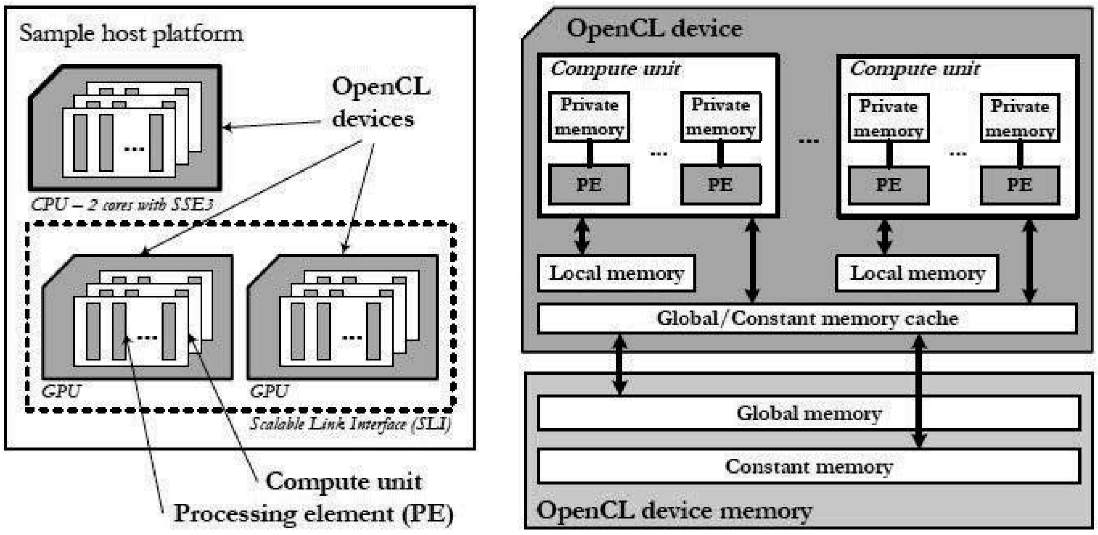


Figure 2: OpenCL device organization

Programmers can access heterogeneous processors (CPUs or GPUs) in the same way, by using the OpenCL API. Source programs are written in specialized language (restricted version of ISO C99, enhanced with some parallel constructs). The advantage of using ATI Stream SDK is that ATI software embraces open-platform standards. Programmable GPU computing devices execute stream kernels (or simply kernels). These devices can execute non-graphics functions using a data-parallel programming model that maps executions onto computing units. In this programming model, arrays of input data elements stored in memory are accessed by a number of computing units.

According to [1], each instance of a kernel running on a computing unit is called a work-item. A specified rectangular region of the output buffer to which work-items are mapped is known as the n-dimensional index space, called an NDRange. The GPU schedules the range of work-items onto a group of stream cores, until all work-items have been processed. Subsequent kernels then can be executed, until the application is completed. OpenCL utilizes four memory domains: private, local, global, and constant, as depicted in the Figure 2. Further within this work, the mentioned ATI SDK is considered.

2.2 Using a GPU for Computer Emulation

Stream processing as a programming model is very suitable for tasks of computational nature. According to [12], I/O (interactive) tasks do not profit performance improvement so much. Therefore within GPUs, the performance improvement is achieved by minimizing the amount of input and output operations and trying to spend as much time as it is possible by calculations.

So the question of high importance here is: What kind of tasks the CPU of computer we want to emulate performs? The compute-bound or interactive-bound processes have better performance, if the computer is adapted for particular kind of work, respectively.

GPUs are adapted for computational tasks, which are well parallelized in the meaning of data-parallelism. Then it can be stated that (using GPUs) it is worth to emulate only compute-bound systems with a tendency of good parallelization. Examples of such systems can be found: data-flow architectures, co-processors, or general-purpose processors about that we actually know they will solve compute-bound tasks. Examples of compute-bound and well-parallelizable tasks include: video/image/sound processing, some physical model computations, cryptographic tasks, neural network simulation, etc. Computers that best fit for performing such tasks are parallel computers. Emulation of those computers would be usually very slow in non-parallel environment, depending on the target/host performance ratio (when it is more than 1). Therefore, it can be assumed that using better non-parallel emulation technique will not help to improve the performance too much.

We can theoretically compute the upper boundary of performance improvement, using GPU as a host for emulation. Let's mark the emulation performance of multi processor (N processor) system on a sequential (single processor) host computer as P_e $[MIPS]$. The nature of stream processing allows emulating each target processor by independent kernel, working in parallel. If the performance of host processor (the P_e) is nearly the same as the performance of single stream processor inside the GPU (let's mark it as P_s), the upper boundary of resulting performance improvement is $N \cdot P_s \doteq N \cdot P_e$. However, it is obvious that the performance improvement using GPU as a host will be achieved also when the single processor host performance $P_e < N \cdot P_s$.

Other issues are related to GPU computational capabilities. First problem is to answer the question if it is possible to emulate each CPU instruction on a GPU. CPU instructions can be broken into several groups, dealing with arithmetic operations, logic operations, memory movement, program control, multimedia instructions (for example MMX extension), etc. According to the specification of streaming languages for GPU (e.g. OpenCL), the GPU is able to execute any of non-I/O instruction directly by some algorithm. The problem will arise when an instruction sends or receives data outside of GPU, as it is not possible to handle the communication of GPU-devices directly. Therefore, the emulator must have a distributed nature, where all non-I/O instructions would be executed by GPU and the rest must be passed to host CPU. The more instructions are passed to CPU, the more emulator performance is lost, due to the communication overhead. Therefore, we can expect performance improvement mainly in emulation of tasks with computational nature.

3. Distributed Emulation

Within a single-platform computer emulator, all target computer components are implemented as software modules or parts (e.g. CPU, memory and I/O devices) that communicate with each other in a sense of procedure calls. Physically, they are all located in the host operating memory, so the physical communication is realized only between host CPU and host operating memory. When we intend to utilize a GPU as another source of computational power, it is necessary to physically separate the emulator modules. In this point, the distributed nature is introduced. All compute-bound components will be executed on a GPU and the I/O bound components (including I/O instructions of emulated CPU) on the host CPU.

3.1 Emulator Organization

The structure of such distributed emulator is depicted in Figure 3. The emulator consists of two main parts which communicate in the distributed manner. The idea is that GPU would execute the CPU emulator that would be implemented as a stream kernel. The host part will act like a scheduler or dispatcher, that would *catch* and *delegate* events either from the target CPU (physically from host GPU), or from other target (emulated) I/O devices (physically from host CPU).

As can be seen in the Figure 3, compute-bound target CPU and operating memory are located in the host GPU. The I/O devices emulators are considered to be I/O-bound by default, and are located in the host CPU. The communication between host CPU and GPU therefore represents the communication between target CPU and I/O devices.

The physical transmission medium (further a channel) between CPU and GPU is usually a PCI Express (PCIe) channel. This channel connects host CPU, host system memory and peripheral

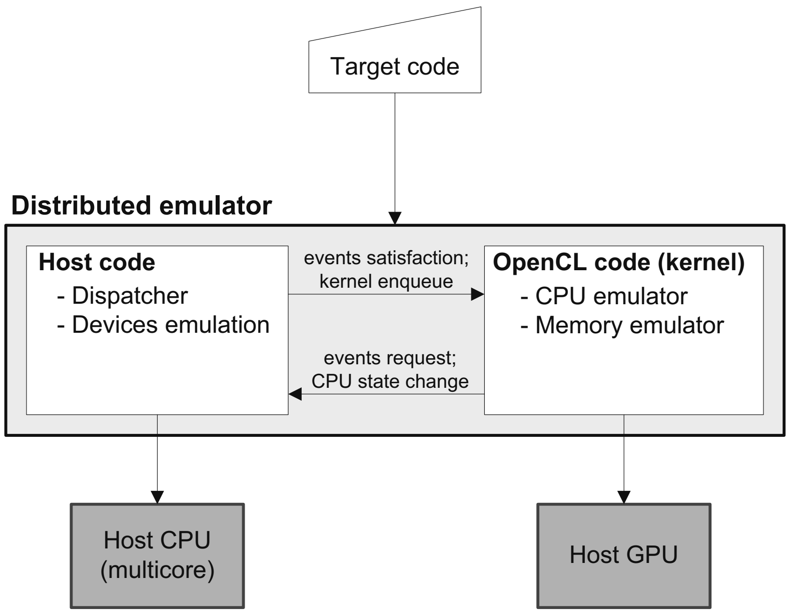


Figure 3: Structure of distributed computer emulator using GPGPU

devices. Transfers from host CPU to GPU are done either by the command processor, or by the DMA engine. The GPU can read and write host operating memory through kernel instructions over the PCIe bus, but other peripheral devices that are connected to PCIe bus cannot be accessed from kernels. Transfer performance is CPU and chipset dependent.

3.2 Event Handling

Even the emulated CPU would be compute-bound, generally it can need some interaction. The interaction can consist, for example, of a reaction to an external event (e.g. device interrupt request). CPU may also ask for service if it needs to send the output data. All the interaction must go through the host-part of the emulator and therefore, some communication mechanism must be provided.

There are more options in CPU-device communication, such as:

- Polling, or busy waiting. In this case, CPU waits for the device, until it is ready to be serviced. The device cannot inform the CPU if it is ready, and therefore the CPU must do the checking manually in wait-loops. If the CPU found a task that can be performed by the device, it waits for the device until it is ready and then sends the data to the device for processing.
- Interrupt requests (IRQs). In this case, devices ask for attention of CPU. They set special signal on system bus that informs the CPU about the service request. In higher abstraction, they can be understood as events that are caught by host CPU (using spooling). The CPU reacts to these requests by postponing the instruction flow just executed and it starts to execute the service procedure. An advantage here is the fact, that the CPU does not have to wait for the device and can do another work independently. Polling may be still possible, as an alternative.

In the first model (polling), only the CPU can request devices. In distributed emulator, target devices implemented in host CPU wait for request from the target CPU, implemented as stream kernel in GPU. A dispatcher is needed that provides message passing from GPU to host CPU and back. In the second model (IRQs) devices also can request the CPU for service. In that case, devices send the request and data to dispatcher, which stores them into a FIFO queue. Independently, the dispatcher should inform the GPU if data are ready. Example situation is depicted in Figure 4.

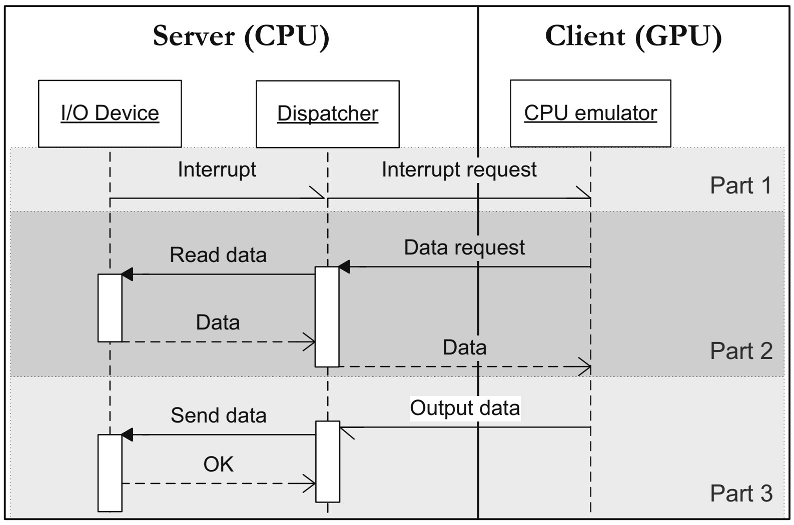


Figure 4: The example communication between host CPU and GPU – events

In the upper part of the figure, a device asks for a service, by generating an interrupt request. The interrupt request is, however, virtual, because it comes from emulated device. The host-part of the emulator catches the event and processes it. If the interrupt needs to be serviced by the emulated CPU, the interrupt request, along with additional information, is 'sent' to the GPU.

However, there is one problem: it is impossible to pass data to a running kernel in a GPU (using any of the present streaming languages). The solution therefore is to use some other communication mechanism. The OpenCL allows using shared memory between the host and GPU, located in the host operating memory. As it was mentioned earlier, the PCIe bus connects a GPU also with the system memory and the OpenCL API allows accessing it both from host and running kernel. So, the 'send' operation from host CPU into GPU means to change some specific data in shared memory by the dispatcher (the 'send' operation in another direction would be performed in the same manner, but the GPU will perform a change of the shared data). Therefore, both GPU kernel and dispatcher must watch the shared memory for changes either in infinite loop using another thread, or in fixed time intervals within the same thread. When GPU kernel encounters the change, the CPU emulator immediately interrupts its execution flow and handles the interrupt in a specified way. If the dispatcher encounters the change, according to the negotiated meaning of the data it performs the specified action (e.g. send data to an output device).

After interrupt request generation, the dispatcher ensures the gathering of required data for the emulated CPU satisfaction. Another component collects the data and sends it to the dispatcher. The communication between dispatcher and other emulated components is usually realized by procedure calls, and data from shared memory are passed as parameters to the procedures. It is possible because both the dispatcher and other target devices are located in host operating memory and executed by host CPU. The dispatcher finally sends the requested data back into the GPU using the shared memory.

The last part of the diagram (Figure 4) is showing the situation, where emulated CPU wants to send data to an output device. The device should confirm that the data sent were accepted correctly.

4. Implementation Considerations

Now it is the time to choose the implementation language used for GPU-part of emulator implementation. Section 2.1 describes available options. For further work, we chose the OpenCL. The main reason is, that it is an open standard and can be used for both CPU and GPU programming using the same API. Moreover, OpenCL is supported by many vendors and free SDK is also available.

4.1 Emulation Techniques

GPGPU is a technique that enables general-purpose computation on graphics cards. It uses a shader language as the lowest-level language (machine language). However, shader languages are mostly enclosed (their specification is not public) and they differ among GPU models. The access is done directly via drivers or higher level streaming languages (e.g. OpenCL, CUDA, DirectCompute; older GPGPU approaches include also OpenGL and DirectX).

The portability problems connected with the enclosed shader languages specifications make use of other emulation technique than interpretation impossible. Another critical problem is, that stream kernels, within all presently available streaming languages, cannot use function pointers. Therefore, only decode-and-dispatch interpreters are possible to construct (instead of faster threaded code interpreters) [21].

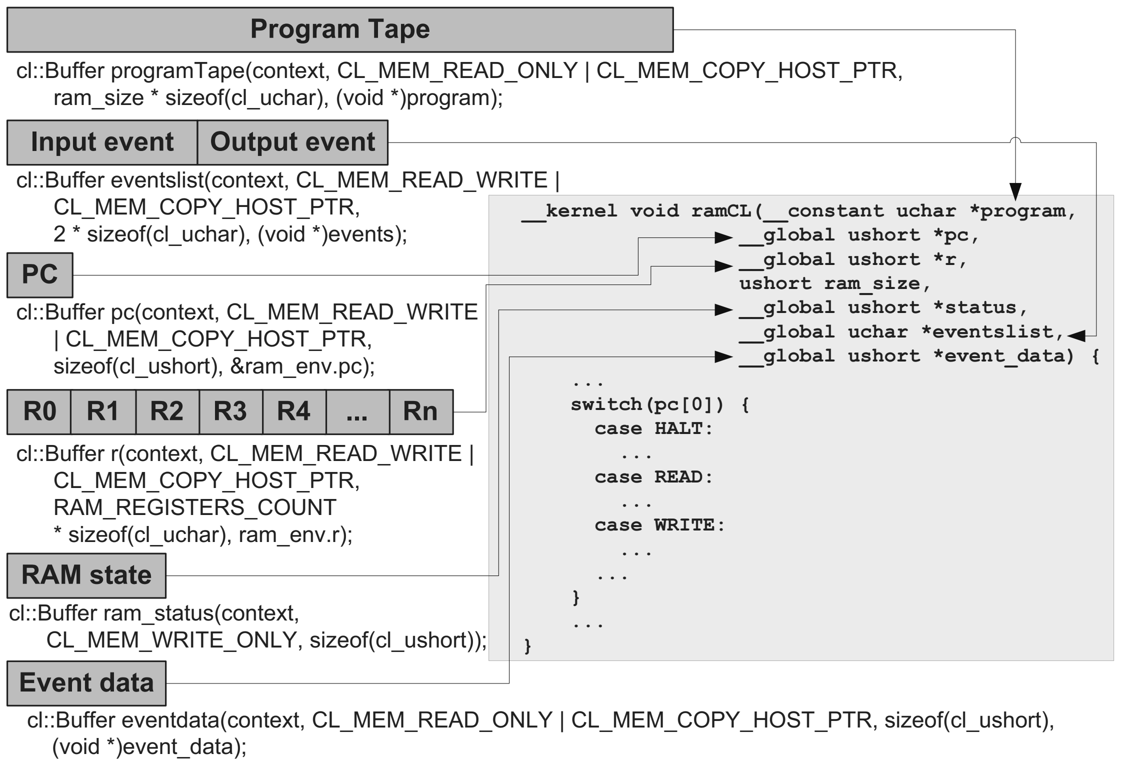
4.2 Issues Connected with Implementation

Main issues of the implementation of distributed CPU emulator include:

- Representation of target CPU structure (e.g. registers, flags, etc.);
- Preservation of correct instruction timing; i.e. making the CPU real-time;
- Consideration of other CPU-related components (e.g. co-processors, operating memory) implementation in OpenCL;
- Design of communication mechanism used to information exchange between CPU and devices (e.g. signals, interrupts, input/output instructions, etc. through shared memory).

Target CPU structure includes all the real storage structures that are visible for programmers. It is the organization structure of the Instruction Set Architecture (ISA) of the CPU, like registers or flags. Besides that, the structure includes other meta-information specific for the implementation, such as interrupt register, processor running state, or profiling information. In single-platform emulators, all items are implemented as single variables or array of variables. The issue here is to find the appropriate array representation of the structure for the stream kernel parameter (kernels don't accept structures as parameters), and to decide what parts of the structure should be held in shared memory (accessible by host CPU and GPU concurrently), instead of private memory within the GPU.

Generally, it is possible to encode each data type into an array of variables of another data type. Stream kernels accept arrays of variables of the same data type, but do not accept structures, which are collections of variables of various data types. The best case seems to be the use of one general data type for all items of the structure, and its implementation as a single array (for example smaller integers can be implemented as larger integers, but not in the other way). A reason for trying to store all-in-one is transfer performance. Each transfer brings some communication overhead before and after the transfer. Now we will call the *context* an instance of the CPU structure in a moment of time (the variables will be modified during the emulation). If the CPU emulator would support debugging, the context GPU-host transfers must be performed very often, in the most precise case after each executed instruction. The example of such structure representation is shown in Figure 5.



```
cl::Buffer programTape(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    ram_size * sizeof(cl_uchar), (void *)program);

cl::Buffer eventslist(context, CL_MEM_READ_WRITE |
    CL_MEM_COPY_HOST_PTR,
    2 * sizeof(cl_uchar), (void *)events);

cl::Buffer pc(context, CL_MEM_READ_WRITE
    | CL_MEM_COPY_HOST_PTR,
    sizeof(cl_ushort), &ram_env.pc);

cl::Buffer r(context, CL_MEM_READ_WRITE |
    CL_MEM_COPY_HOST_PTR,
    RAM_REGISTERS_COUNT
    * sizeof(cl_uchar), ram_env.r);

cl::Buffer ram_status(context,
    CL_MEM_WRITE_ONLY, sizeof(cl_ushort));

cl::Buffer eventdata(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_ushort),
    (void *)event_data);
```

```
__kernel void ramCL(__constant uchar *program,
                    __global ushort *pc,
                    __global ushort *r,
                    ushort ram_size,
                    __global ushort *status,
                    __global uchar *eventslist,
                    __global ushort *event_data) {
    ...
    switch(pc[0]) {
      case HALT:
        ...
      case READ:
        ...
      case WRITE:
        ...
      ...
    }
    ...
}
```

Figure 5: The CPU context used in RAM emulator

We decided to store the context in GPU's global memory (according to OpenCL GPU architecture). Another option is to use (faster) local memory. However, host-GPU transfer goes through global memory and therefore it depends on how often the context is transferred between host CPU and GPU. Also, when the emulator uses several stream kernels that would not be logically grouped into the same work-group, they cannot share data stored in local memory.

5. Case Study: RAM Machine Emulator

This section gives an overview of RAM machine emulator implementation, using OpenCL. RAM is an abbreviation for Random Access Machine. It is an abstract machine (theoretical model of computation) used primarily for computational complexity analysis [8], [11].

Within a single-platform emulation, the RAM emulator is very easy to implement. Dynamic project [9] by the first author of the paper is an experimental software project for RAM emulation, intended for testing of various emulation techniques. Within this project, interpretation and dynamic translation were tested. The results achieved were 97 MIPS using a dynamic translation technique and about 9.2 MIPS using interpretation respectively. For performance measurements a PC with Intel Core2 Duo at 1.66Ghz and 2GB RAM was used. Further enhancement of the project is RAM emulation using GPGPU.

5.1 Context of the RAM Machine

The RAM machine uses arbitrary number of internal registers, represented by the register tape. The registers within the RAM model are cells that can contain symbols representing values. In such way, additional three tapes are defined – input tape, output tape and program tape. All the tapes, except the program tape are left-bounded and unbounded on the right. The program tape is bounded on both sides. Within the RAM machine as a theoretical model, data types of the tape values are not considered. This is the first issue that must be solved in the implementation, because data need to be represented by some data type. Another, more important issue is that RAM machine has (except the program tape) infinite tapes and this is the feature, which is not possible to implement. The proposed emulator therefore does not represent complete RAM machine, but only imitates its functionality with the limited structure.

Architecture of RAM machine is of Harvard type. It means that separate memories for instructions and data are used. The read-write memory storing the computation data is the register tape. Another tapes represent the I/O devices – inputs are read only from the input tape and outputs are written onto the output tape. No other devices are used within the machine.

The implementation of the RAM machine would represent a reference point for implementing any other real computer. By implementing as simple machine as is RAM, it would be possible to describe the model of communication between host CPU and kernel in GPU clearly, together with GPGPU-based computer emulation technique used.

The context should consist of a processor-like structure. The RAM machine context (Figure 5) stores:

- Program tape;
- Program counter (PC);
- Registers tape;
- Events list;
- Event data buffer;
- State of the machine.

The other components are considered as external memories or I/O devices. The mechanism for transferring the context from host CPU to GPU and back is realized using the OpenCL API.

5.2 Memory Implementation

For effective memory implementation, it is better to place the emulated operating memory inside the GPU's global memory. This allows local accesses to memory, which do not cause the performance issues. Emulator uses the memory as if it was executed on host CPU, preserving the memory format. However, this is not always possible. If emulated operating memory is larger than available global memory, or if a virtual memory is supported, the memory cannot be implemented in global memory as a whole. This case is more complex mainly for performance issues, which are connected with data transfers. One solution would be to use GPU-CPU shared memory (as it is used for transfer of events, but it would be probably much larger), with MMU implementation on the host CPU.

Implementation of RAM machine involves in fact four memories – the tapes. If we consider input and output tapes will represent I/O devices, next two memories remain for the implementation here. For the sake of simplicity, we will limit the program tape and registers tape in order to fit into global memory. The limit for registers tape will be 100 registers and the program tape will be limited by the program size.

The memories are implemented in the same way as the context. At the beginning, special buffers are created using the OpenCL API (`clCreateBuffer` function). The parameters of the function pass the RAM program and initial values of registers respectively into the buffers. Note that the buffers are stored inside the system operating memory and they must be transferred into GPU.

5.3 I/O Communication

Both input and output tapes are considered to represent input and output devices, respectively. Therefore, they are not stored in memory like common buffers, but interaction is simulated using events transfer, as it was described in section 3.2. However, CPU-GPU shared memory is not used for the event transfer and communication. Another mechanism is used here, based on polling, as it is the easiest model of communication between CPU and devices. The example in the Figure 4 would represent polling, if the interrupt request from a device would be omitted.

Events are represented by two memory objects, or special buffers, similar to the buffers for storing the context and registers. The first one represents events list – it is an array of all available events that can happen. Each array index represents a code of the event. The second buffer represents the data associated with the communication. In the RAM emulator, there are only two kinds of events – input request event and output request event. The *input request event* (or IRE) occurs by executing `READ` instruction within the emulator. The *output request event* (or ORE) occurs when `WRITE` instruction is executed by the emulator. Admissible values of the event items in the events list are given in Table 1.

Value	**Description**
0	No event
1	Event request is active
2	Event is satisfied by dispatcher

Table 1: Event values description

The task of the host CPU-based dispatcher is to catch the event, perform the action related to the event, and return a result back to the GPU-based part of emulator. However, it is impossible to pass data into the running kernel. Therefore, the dispatcher must do the following:

1. Wait for kernel to quit;
2. Transfer events list into dispatcher;
3. Check for event requests;
4. Satisfy events by:
    - `READ` instructions: reading the input from the keyboard and store it to the event data buffer;
    - `WRITE` instructions: transferring event data buffer from the GPU into temporary variable and then passing it to the output tape.
5. Set involved items in events list to the value 2 (satisfied);
6. Transfer the events list back to the GPU, including the event data buffer in case of `READ` instructions;
7. Execute the kernel again.

The dispatcher algorithm is presented in Figure 6. When the kernel quits, all the data are still active in the GPU, so it is up to the programmer what data will be transferred to the host CPU. Then the repeated kernel execution can start without transferring all the data. Only some of them may require update (e.g. the events list and event data buffer), so the execution can begin very quickly.

Within the kernel, when the emulator encounters `READ` or `WRITE` instruction, it sets the appropriate value in the events list to 1, and possibly (in the case of `WRITE` instruction) fills the event data buffer. Then, program counter must be decremented (generally set to the previous instruction), in order to point to the `READ` or `WRITE` instruction again. The final step is that the kernel will quit.

The reason why the program counter must be decremented is connected with the next kernel execution. When it is executed again by the dispatcher, the I/O instruction is executed again and it checks if the event is satisfied. If it is satisfied, it performs specified operation and continues by executing the next instruction. If it is not, event request is created and the situation repeats. In the Figure 7, the OpenCL implementation of the two I/O instructions is shown.

```
...
p_output = 0;
p_input = 0;

while (true) {
    // run the emulator on GPU
    executeRAMkernel(&event);

    // wait for kernel finish
    event.wait();

    // read eventslist memory to the host
    eventslist = readEventsListFromGPU();

    // read RAM state
    state = readRAMstateFromGPU();

    // test if emulator was really halted
    if (state != RUNNING)
        break;

    if (eventslist[0] == 1) { // input request?
        // read input
        data = input_tape[p_input++];

        // put the input back to the global memory
        writeDataToGPU(data);

        // update eventslist in the global memory
        eventslist[0] = 2; // satisfied
    }
    if (eventslist[1] == 1) { // output request?
        // read data for output
        data = readDataFromGPU();

        // write output
        output_tape[p_output++] = data;

        // update eventslist in the global memory
        eventslist[1] = 2; // satisfied
    }
    writeEventsListToGPU(eventslist);
}
...
```

Figure 6: Dispatcher algorithm in GPU-based emulator

6. Conclusions

Reasons for using simulators and emulators are manifold as it was described within the paper introduction. The advantages of full system emulation (or simulation) are several and we only give some of them here. Software development can take place before the hardware is ready, so the verification/validation process can proceed faster. Validation is less costly and faster because many engineers can run validation tests on their PCs instead of sharing few HW prototypes. Some tasks can be done using simulation that can hardly be done with the hardware, like verifying correct hardware initialization, simulating defective hardware, internal observations, etc. And the last, but not the least advantage - simulation tools can be connected with formal methods [7], [20] to increase the confidence in product.

```
switch (pc[0]++) {
    ...
    case 1: /* READ i */
        // test if the input event has been
        // already satisfied
        if (eventslist[0] == 2) {
          r[program[pc[0]++]] = event_data[0];
          eventslist[0] = 0;
        } else {
          // wait for the event = return one
          // instruction back, request event
          // and exit
          pc[0]--;
          eventslist[0] = 1;
          return;
        }
        break;
    case 4: /* WRITE i */
        if (eventslist[1] == 2) {
          eventslist[1] = 0;
        } else {
          // wait for the event = return one
          // instruction back, request event
          // and exit
          pc[0]--;
          eventslist[1] = 1;
          event_data[0] = r[program[pc[0]++]];
          return;
        }
        break;
    ...
```

Figure 7: OpenCL implementation of READ and WRITE RAM instructions

Stream processing and GPGPU brought a new potential of making high-performance software also for computers widely available – standard PCs. We believe the GPGPU programming model offers valuable properties also for emulator/simulator development. GPGPU supports it by limited form of parallelism, which leads to applications divided into two parts – host part and client part. Host part is implemented in any of standard programming languages (like C or C++) and it is executed on CPU. Client part is written in specialized streaming language, providing access to powerful parallel hardware. The client part is executed on GPU, which disburdens the CPU from computational work that can be parallelized.

This work points at utilizing this new approach in development of computer emulators (simulators) that will run as client parts of the application, on a GPU. However, some restrictions must be taken into account at the design of the emulator, given by the restrictions of a streaming language. Considering these restrictions we proposed the RAM machine context, memory implementation and I/O communication mechanism. Practical results of our work are summarized in chapter 5 of the paper, containing a description of RAM machine emulator implementation, based on ideas presented within the paper. Complete source code of the emulator can be found in [9].

We have got a lot of useful knowledge when designing, implementing and testing the RAM machine emulator described within the paper. But when it comes to performance evaluation of the emulator, we can conclude from the measured values (our testing configuration included Intel Core i3 CPU at 3.10 GHz and AMD Radeon HD 6570 display adapter, running Stream SDK v2.3), that the emulator itself ran much faster on the CPU than on the GPU device. The reasons for such results include tasks like transferring the kernel code into GPU, transferring buffers with data, etc.

But the main reason is the fact, that emulator of the sequential RAM machine was implemented. So in this case, the main potential (parallel processing) of the GPGPU was not fully utilized. Even though the saved CPU time can be used for performing other tasks and overall performance increased. Better results in this respect would be obtained by emulating the parallel models, e.g. the Parallel RAM (PRAM) model [5], [4] so the future work would include implementation of PRAM model based on extension of existing RAM model implementation.

We believe despite of all the issues, it is worth to study and discover new possibilities of this approach. There are still other open issues that were not analyzed deeply enough and they require further research. The most important of them are:

- Preservation of target CPU speed – cycle timing accuracy;
- GPU-based device emulation;
- Optimization of communication between devices and CPU;
- Performance evaluation of GPU-based emulator.

Acknowledgment

This work is the result of the project implementation: Development of the Center of Information and Communication Technologies for Knowledge Systems (ITMS project code: 26220120030) supported by the Research & Development Operational Program funded by the ERDF.

References

[1] *AMD Accelerated Parallel Processing OpenCL*, Advanced Micro Devices (AMD), 2010. [Online]. Available: http://developer.amd.com/gpu/ATIStreamSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[2] Duncan, R., “A survey of parallel computer architectures,” *Computer*, vol. 23, pp. 5–16, February 1990. [Online]. Available: http://portal.acm.org/citation.cfm?id=78692.78693

[3] Flynn, M. J., “Very high-speed computing systems,” in *Proceedings of the IEEE*, vol. 54, 1966, pp. 1901–1909.

[4] Formella, A., Grn, T., Keller, J., Paul, W., and Rauber, T., “Scientific applications on the sb-pram,” in *Proceedings of the International Conference 'Multiscale Phenomena and Their Simulation'*, Karsch, F., Monien, B., and Satz, H., Eds. Bielefeld: World Scientific Publishers, 1997. [Online]. Available: http://www-wjp.cs.uni-saarland.de/publikationen/FGKPR97.pdf

[5] Fortune, S. and Wyllie, J., “Parallelism in random access machines,” in *Proceedings of the tenth annual ACM symposium on Theory of computing*, ser. STOC '78. New York, NY, USA: ACM, 1978, pp. 114–118. [Online]. Available: http://doi.acm.org/10.1145/800133.804339

[6] Harris, M., “General-purpose computation on graphics hardware,” 2011. [Online]. Available: http://gpgpu.org/

[7] Hudák, S., *Reachability Analysis of Systems Based on Petri Nets*, 1st ed. Elfa, 1999.

[8] Hudák, S., *Machine oriented languages (in Slovak).* Faculty of Electrical Engineering, Technical University of Košice, 2003.

[9] Jakubčo, P., “Dynamic software project,” 2011. [Online]. Available: https://github.com/vbmacher/dynamic

[10] Kapasi, U., Dally, W. J., Rixner, S., Owens, J. D., and Khailany, B., "The imagine stream processor," in *Proceedings 2002 IEEE International Conference on Computer Design*, sep 2002, pp. 282–288.

[11] Kasai, T., "Computational complexity of multitape turing machines and random access machines," *Publications of the Research Institute for Mathematical Sciences*, vol. 13, no. 2, pp. 469–496, 1977. [Online]. Available: http://www.ems-ph.org/journals/show_pdf.php?issn=0034-5318&vol=13&iss=2&rank=5

[12] Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Owens, J. D., and Towles, B., "Exploring the vlsi scalability of stream processors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03. Washington, DC, USA: IEEE Computer Society, 2003, p. 153. [Online]. Available: http://portal.acm.org/citation.cfm?id=822080.822826

[13] MSDN Channel9, "Direct compute lecture series," 2010. [Online]. Available: http://channel9.msdn.com/tags/DirectCompute-Lecture-Series/

[14] Munshi, A., *The OpenCL Specification*, Khronos OpenCL Working Group, 2011, revision 44. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[15] *NVIDIA's Next Generation CUDA^{TM}Computer Architecture: Fermi^{TM}*, NVIDIA, 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[16] *NVIDIA OpenCL programming guide for the CUDA architecture. Version 3.2*, NVIDIA, 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Programming_Guide.pdf

[17] Patterson, D. A. and Henessy, J. L., *Computer organisation and design*, 4th ed. Morgan Kaufmann, 2008.

[18] Rixner, S., *Stream Processing Architecture*. Kluwer Academic Publishers, 2002.

[19] Shebanow, M. C., "Pervasive massively multithreaded gpu processors," in *Proceedings of the 6th ACM conference on Computing frontiers*, ser. CF '09. New York, NY, USA: ACM, 2009, pp. 227–227. [Online]. Available: http://doi.acm.org/10.1145/1531743.1531745

[20] Slodičák, V., "Some useful structures for categorical approach for program behavior," *Journal of Information and Organizational Sciences*, vol. 35, no. 1, pp. 93–103, 2011. [Online]. Available: http://jios.foi.hr/

[21] Smith, J. and Nair, R., *Virtual Machines: Versatile Platforms for Systems and Processes*, 1st ed. Morgan Kaufmann, 2005.

[22] Wolfram Research, Inc., "Opencllink guide," 2010. [Online]. Available: http://reference.wolfram.com/mathematica/OpenCLLink/guide/OpenCLLink.html

[23] Zhao, Y., "Streaming processor," 2011, lecture of course CS 6/79995: GPU computing, Kent State University. [Online]. Available: http://www.cs.kent.edu/~zhao/gpu/lectures/StreamProcessor.pdf