Podpora CUDA kódu v multiagentní platformě JADE

CUDA code support in Multiagent platform JADE
Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 06.05.2011

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 06.05.2011
I would like to thank my supervisor Ing. Hussam Abdulla for his support and guidance in my work.
Abstrakt
Cílem této diplomové práce je implementace podpory CUDA kódu v Multiagentním frameworku JADE. Agenti jsou schopni posílat PTX kód nebo CUDA zdrojový kód prostřednictvím ACL zpráv a tento kód spouštět na GPU. Kromě samotné integrace podpory CUDA kódu je rovněž cílem této diplomové práce měření výkonnosti takového řešení a její vyhodnocení.

Klíčová slova: diplomová práce, multiagentní systémy, JADE, CUDA, Java, ACL zprávy, PTX kód

Abstract
The main goal of this Master’s thesis is to implement CUDA code support in the Multiagent framework JADE. The agents are able to send a PTX code or CUDA source code via ACL messages and run the code on the GPU. Aside from integrating CUDA support, this Master’s thesis is also focused on measuring performance of this solution and evaluating the performance results.

Keywords: master thesis, multiagent system, JADE, CUDA, Java, ACL messages, PTX code
List of abbreviations

GPU - Graphics Processing Unit
CUDA - Compute Unified Device Architecture
CPU - Central Processing Unit
CUBLAS - Compute Unified Basic Linear Algebra Subprograms
CUFFT - Compute Unified Fast Fourier Transform
SIMT - Single Instruction Multiple Threads
PTX - Parallel Thread Execution
GUI - Graphical User Interface
MAS - Multi Agent System
AMS - Agent Management System
DF - Directory Facilitator
Contents

1 Introduction 4

2 Parallel computing on the GPU 5
   2.1 Massively parallel processing power on the GPU 5
   2.2 Introduction to the CUDA technology 5
      2.2.1 The CUDA Architecture overview 5
      2.2.2 Compute capability 6
      2.2.3 Kernel 6
      2.2.4 Threads, blocks and grid 7
      2.2.5 Memory and coalesced access 8
      2.2.6 Global memory 9
      2.2.7 Shared memory 10
      2.2.8 Local memory 10
      2.2.9 Registers 10
      2.2.10 Constant cache 10
   2.3 CUDA roadmap 11

3 Development with CUDA SDK 12
   3.1 Introduction to CUDA Toolkit and GPU Computing SDK 12
   3.2 Installation on Unix systems 12
   3.3 CUDA C API and CUDA C source code example 13
   3.4 Compilation and the binary output analysis 14
   3.5 Debugging and Compute Visual Profiler 16

4 Application area of CUDA 18

5 JCuda wrapper 19
   5.1 Installation 19
   5.2 Development with JCuda 20
   5.3 Unit testing 22

6 Introduction to Multiagent system 24
   6.1 Basic concept of MAS 24
   6.2 MAS Frameworks 24
   6.3 Agents 24
   6.4 Ontology 25
   6.5 Performative 26
   6.6 ACL Messages 27

7 JADE 28
   7.1 JADE Architecture 28
   7.2 Installation 29
   7.3 JADE Agent example 29
7.4 JADE Ontology ............................................. 30

8 Integration of CUDA kernel code transfer using ACL messages and running of this code 35
8.1 The utilization of CUDA within MAS .............................. 35
8.2 Introduction to a CudaAgent ........................................ 35
8.3 The CudaAgent Implementation .................................. 36
8.4 The CudaAgent ontology and performative ....................... 36
8.5 The CudaAgent methods and data transfer ....................... 38
8.6 Usage of the CudaAgent .......................................... 39
8.7 MAS Example ..................................................... 39

9 Evaluation of performance tests and summary results 41
9.1 CUDA Kernel call ................................................. 41
9.2 Kernel result data within the ACL Message ..................... 42
9.3 Java emulation kernel call ....................................... 42
9.4 Performance tests and summary results ........................ 43

10 Conclusion ...................................................... 45

11 References ........................................................ 46
List of Figures

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The CUDA Architecture</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Grid of thread blocks</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>the nvcc compiler schema</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>The FIPA Ontology</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>The hierarchical ontology</td>
<td>26</td>
</tr>
<tr>
<td>6</td>
<td>The JADE architecture</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>The JADE ontology classes</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>The cuda kernel transfer</td>
<td>36</td>
</tr>
<tr>
<td>9</td>
<td>time spent on the kernel call</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>time spent on received kernel result</td>
<td>44</td>
</tr>
</tbody>
</table>
1 Introduction

The work presented in this diploma thesis tries to implement special Agent in Multi Agent System that it can parallel process a binary code on the CUDA Technology and provides this service other agent within Multi Agent System.

There exist several science branches such as finance, medicine or data mining that manipulate with huge data and difficult algorithms and therefore need to process their algorithms quickly. The CUDA Technology based on the parallelism provides a way how we can process this algorithms faster than CPU (up to 400x times faster). The Multi Agent System is a system composed from intelligent agents communicate with each other and solve non-trivial problems.

In this diploma thesis is looking for answer how we can fast solve agent sub-problems in the MAS system that need big amount of memory space and also need long time to process. In other words we want to implement Agent provides capable run compiled algorithms on the GPU. We also define own ontology towards transferring source code or binary code of algorithms to our specialised agent able to run this code within CUDA Technology.

The second task is measure performance of this solution, especially performance of run the code and performance of transfer a result data back to the agent. We compare the performance results of our solution with normal CPU solution without parallelism. We demonstrate our agent using Java platform, CUDA Technology and JADE framework.
2 Parallel computing on the GPU

In this chapter we will discuss Parallel computing on the GPU, i.e. massively parallel processing power, hardware implementation and programming model of the CUDA technology. At the end of this chapter, JCUDA wrapper will be discussed also.

2.1 Massively parallel processing power on the GPU

Once GPU hardware was designed for computer graphics. Computer graphics have required better rendering speed over time. Fortunately there are many tasks that can be processed as parallel in computer graphics. Therefore, the GPU requires more power because of a greater number of processors and cores. On the other hand this power might have been used for parallel processing in general. The earliest computer graphics API such as OpenGL and Direct3D allow modification graphics pipeline through vertex and fragment shaders. Vertex and fragment shaders became a very popular "tool" for developing a graphics engine. They made it easy to programming our lighting model or some effects such as refraction, night vision or motion blur in real time. Today’s GPU hardware also offers also a programming interface that can be used for massively parallel processing. The modern GPU brings more massively processing power than today’s multi-core CPU (scaling from 2 to 8 cores) because of a large number of cores (one GPU can have up to 580 cores in total). Massively parallel processing power on the GPU brings new possibilities unrealizable never before such as computation in physics, math, medicine and cryptography. Therefore, graphics vendors NVIDIA and AMD/ATI support accessible programming interfaces. These programming interfaces use the language C in most cases. There have been two main programming interfaces dedicated to massively parallel processing on the modern GPU. One of them is an open standard OpenCL and the other is a proprietary CUDA technology (an acronym for Compute Unified Device Architecture). The CUDA technology (earlier interface than the OpenCL on the market) has provided robust software libraries and toolkits like CUBLAS, CUFFT and the Visual Profiler in comparison with the standard OpenCL. Therefore, more details about the CUDA technology will be discussed in the next paragraphs.

2.2 Introduction to the CUDA technology

In this section CUDA technology will be discussed. We will explain the basic concept of the CUDA environment including a kernel, threads and blocks, memory spaces and coalesced access to a memory.

2.2.1 The CUDA Architecture overview

As we can see in the figure 1. There are several parts in CUDA technology. At the bottom of CUDA technology we can see CUDA Parallel Compute Engine. The second part of the CUDA support for hardware initialization, configuration and other internal processes. The third part of CUDA is a user-mode CUDA driver. The CUDA driver provides
a device-level API for developers. Part four is a PTX instruction set architecture (ISA) for parallel computing kernels and functions (we will discuss kernels in section 2.2.3). Another part marks color blue implies possible to develop parallel applications with other API than standard C CUDA API such as DirectX Compute and OpenCL. In other words these API share the same CUDA low-level resources. At the top of the picture are parallel applications calling one of the APIs referred to above.

Developers can use one of two different programming interfaces. There is a device-level programming interface and a language integration programming interface. The device-level programming interface is where the application uses DirectX Compute, OpenCL or the CUDA Driver API directly to configure the GPU, launch compute kernels, and read back results. On the other side of the coin we have the language integration programming interface, in which an application uses C Runtime for CUDA and developers use a small set of extensions to indicate which compute functions should be performed on the GPU instead of on the CPU.

2.2.2 Compute capability

There are several compute capabilities in CUDA technology. The compute capability is defined by majority and minority versions of CUDA on the specified device. Same majority version means same core architecture [8]. The minor revision number corresponds to smaller improvements including new features. In this section only 2.X compute capability based on Fermi architecture will be discussed.

Fermi architecture brings several new core features and makes CUDA easy to use in comparison with 1.X compute capability.

2.2.3 Kernel

The kernel is a special C function in CUDA C environment. The kernel, when called in a program, is executed N times on the GPU depending on the number of threads. The parallel execution is a key advantage of CUDA C in comparison with the standard C. Every thread is independent of the other threads, in which the kernel is concurrently called.

The kernel is defined using keyword __global__ before C function declaration as we can see in listing 1. Each thread has a unique thread ID. There is a built-in variable threadIdx that represents thread ID in the kernel function.

In the place in which a kernel is called a programmer has to define how many threads will be executed. In other words, a programmer defines the execution configuration. The execution configuration has a special syntax “<<<>>>”. This syntax is obviously not for C-like languages. The number of threads and blocks are defined between “<<<" and “>>>”, as we can see in listing 2. The threads in the execution configuration can be specified using a one-dimensional, two-dimensional, or three-dimensional thread index (like a vector). There is a variable threadsPerBlock represents thread index in listing 2. The thread index is formed in a one-dimensional, two-dimensional, or three-dimensional thread block. Each thread has a unique thread ID as we said above. The thread ID (or
built-in variable threadIdx in the kernel) represents the relationship between threads and blocks. In other words for a one-dimensional block the thread ID and threadIdx are the same but for a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is (x + y Dx) and for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dz Dy).

```c
__global__ void VecAdd(float* A, float* B, float* C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Listing 1: Declaration of the kernel

```c
vecAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Listing 2: The execution configuration when is a kernel called

### 2.2.4 Threads, blocks and grid

The variable threadIdx is a natural way across compute matrices or vectors in the programming languages. Therefore, a programmer makes use of built-in variable threadIdx for adding two matrices provided, listing 3. A programmer takes into account the limit of the number of threads and blocks. A thread block may contain up to 1024 threads but each GPU has a different limit. Blocks are organized into one-dimensional or two-dimensional grid of thread blocks. Obviously, the number of thread blocks in a grid depends on the size of the kernel data and the number of processors.

Each thread in a block can cooperate via shared memory and synchronizations. Share memory access strikingly reduces memory bandwith. Blocks are organized into a grid that can be one-dimensional or two-dimensional. The dimension of the grid is represented by the variable numBlocks in listing 3. We can see more clear schema of threads, blocks and grid in figure 2.

As you can see in listing 3 he variable numBlocs (or threadsPerBlock) can be defined like int or dim3. We know that threadIdx represents the thread ID. Similar to the built-in variable threadIdx a programmer can use other built-in variable blockIdx to represent the block ID. A Block ID is a unique block identifier within a grid. And finally, a built-in variable blockDim holds the dimension of a thread block. In conclusion we can look at table 1 describing all built-in kernel variables discussed above.

As mentioned above, the threads within a block can be synchronized. A programmer can do that by calling a built-in function __syncthreads(). This synchronization point anywhere in the kernel means that the threads within a block must wait for each other. We can imagine __syncthreads() like a breakpoint. If each thread within a block reached breakpoint __syncthreads() than the threads within the block are allowed to proceed. In other words the function __syncthreads() provides a simple way how a programmer can coordinate threads within a block.

```
// Kernel definition
```
```c
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
} int main() {
    // Kernel invocation with one block of N * N + 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```


2.2.6 Global memory

Global memory is the slowest memory in the GPU. Global variables should be declared as `__global__`. Also this memory can be allocated with a `cudaMalloc` function from an application. This memory is necessary for storing big amounts of data which are transferred from the host memory (PC) to a device global memory (GPU). Global memory is also available from a host. We must keep in mind that global memory space is not cached. Therefore, a programmer has to know the right access pattern to get maximum memory bandwidth and increase the kernel performance. If a programmer adheres to an access pattern the access to the global memory from a thread is coalesced. Coalesced access brings drastically more kernel performance (minimize the number of bus transactions) than non-coalesced access. A programmer can bring data from global to on-chip memory (i.e. shared memory) in one instruction provided his fetch 32/64/128 bit data is properly aligned.

How does coalesced access work? The multiprocessor executes threads in a group of 32 threads called a warp. Each thread of the warp starts to execute at the same instruction address, but of course, each thread is independent. Coalesced access means coordinated read by a warp. The „Xth“ thread in a half-warp must access the „Xth“ element in a memory block being read. Each element must be aligned to 4, 8 or 16 bytes size. If there

Figure 2: Grid of thread blocks
is a permuted access by threads or if there is a misaligned starting address (not a multiple of 64) that means there is non-coalesced access.

### 2.2.7 Shared memory

Shared memory is fast memory. Shared memory can be used to reduce access to global memory by a thread within a block and increases the application performance. This memory is dedicated to local variables that are accessible within a block. In other words, each thread within a block can access these variables and therefore may be suitable to data exchange within a block. The access to this memory is faster than access to global memory. The variables are stored and the shared memory must be mark as __shared__.

Similar to global memory, shared memory also has a few rules to obtain rich high memory bandwidth and concurrent access. Shared memory is divided into 32 banks. A bank conflict can occur between threads in the first half-warp (first 16 threads in a warp) or second half-warp of the same warp. If a bank conflict does not occur then shared memory access is not realized.

### 2.2.8 Local memory

The local memory space has the same latency and memory bandwidth as global memory because it resides in the device memory. The rules for coalesced access are the same as for global memory. Local memory is always cached and it is accessible only by a thread. Therefore, its lifetime is the same as the thread lifetime. Each variable without qualifiers such as __device__, __shared__ and __constant__ resides on registers. But in some cases a compiler can decide that a variable resides in local memory:

- Arrays for which it cannot determine that they are indexed with constant quantities.
- Large structures or arrays that would consume too much register space.
- Any variable if the kernel uses more registers than available.

### 2.2.9 Registers

Registers are the fastest memory a programmer can use. As we said above, each variable without qualifiers such as __device__, __shared__ and __constant__ resides in registers except for some cases described above. All registers are 32 bit aligned.

### 2.2.10 Constant cache

Constant cache memory resides in the device memory. This memory is also the fastest memory. If a programmer wants to use a variable as a constant then he/she has to mark it with __constant__ specifier. The constant cache is also accesible from a host. For this purpose, a programmer can use cudaMemcpyToSymbol and cudaMemcpyFromSymbol functions. A programmer can also place an array to constant cache memory but can’t reallocate it. Therefore, the size of the array has to be predefined.
2.3 CUDA roadmap

Today’s major architecture is Fermi as we said in Section 2.2.2 on page 6. The next CUDA architecture is Kepler and it will be released in the year 2011. Kepler will be followed by the new architecture Maxwell. Maxwell is planned for the year 2013. Kepler will be based on 28nm process technology.
3 Development with CUDA SDK

In this section, the CUDA software development kit will be discussed, i.e. installing on the Unix like systems, C CUDA API will be further explored including a source code example and also the way a profiler and debugger work will be described.

3.1 Introduction to CUDA Toolkit and GPU Computing SDK

The CUDA Toolkit and the GPU Computing SDK can be obtained from the website \(^1\) of NVIDIA. There are several executable files needed to correctly install this SDK including CUDA drivers and the CUDA toolkit. The newest stable version of GPU Computing SDK is now 3.2. The GPU Computing SDK also provides a number of examples for the CUDA environment, not only for CUDA C runtime but also for OpenCL and provides examples and tutorials, and possibly shows the interoperability between CUDA and Direct3D, and between CUDA with OpenGL, respectively.

3.2 Installation on Unix systems

First, the CUDA driver needs to be installed. If an X window is running then it should be shutdown. When a CUDA driver is installed the CUDA toolkit and possible CUDA Compute SDK can then be installed. All files required for CUDA application development such as. nvcc, cuda-gdb, and examples, are included in the CUDA toolkit. We can see the Installation guide dedicated to a Ubuntu distribution of linux in listing 4.

```bash
# at first install linux headers and other development tools
$ sudo apt-get install build-essential linux-headers-$'(uname -r')
# make backup our X configuration file if something failed.
$ sudo cp /etc/X11/xorg.conf /etc/X11/xorg.conf.original
# reconfigure X server
$ sudo dpkg-reconfigure --phigh xserver-xorg
# remove current (non-cuda capability) nvidia driver
$ sudo apt-get --purge remove $(dpkg -l | grep nvidia | awk '{print $2}')
# edit linux -restricted-modules-common to disable the nvidia driver module installed from standard repository
$ sudo gedit /etc/default/linux-restricted-modules-common
# stop GNOME desktop if it is running
$ sudo /etc/init.d/gdm stop
# run cuda developer driver
$ sudo /dev/driver_3.2_linux_64_260.19.26.run
# after successful installation the cuda driver we can run GNOME again
$ sudo /etc/init.d/gdm start
# install cuda toolkit contains all needed libraries for development
$ sudo /cuda/toolkit_3.2.16_linux_64_ubuntu10.04.run
# also install sample codes
$ sudo /gpusoftware/gpusdk_3.2.16_linux.run
# add path to the cuda sdk directory (nvcc, cuda-gdb)
$ echo "export PATH=/usr/local/cuda/bin:$PATH" >> ~/.bashrc
$ echo "export LD_LIBRARY_PATH=/usr/local/cuda/lib:$LD_LIBRARY_PATH" >> ~/.bashrc
```

\(^1\)http://developer.nvidia.com/object/cuda_3_2_downloads.html
Listing 4: The installation guide to install CUDA driver and CUDA Toolkit on the Ubuntu system

3.3 CUDA C API and CUDA C source code example

In this section more about the CUDA C API concretely used in the CUDA language integration programming interface (discussed in section 2.2.1 on page 5) for managing the GPU and calling a kernel will be described. Listing 5 shows a simple example (and short version) of a CUDA matrixMul. There is a matrix multiplication that is copied two randomly generated large matrices on the GPU and multiplies them via matrixMul kernel. The result is copied back to the program on the host. At first the GPU is chosen on which to run the kernel. Next, there are a few lines with cudaMalloc functions with the allocated required size of the global memory on the GPU for the matrices and returned pointer of allocated memory as like malloc in standard C library. The next lines are represented by copied generated data from the host memory space to the global memory space on the GPU. The next line of source code is called the kernel with the execution configuration that specifies grid and block dimensions. After the kernel proceeds successfully, the program continues to copy the result (which is represented by variable C) back to the host memory space by way of call cudaMemcpy function. Finally, the program has to free the allocated memory on the GPU via cudaFree function. As we said earlier, this listing is only a short version of the matrixMul example that is located in the CUDA Computing SDK directory.

If we want to see every example composing CUDA Computing SDK just type make in the directory $NVIDIA_GPU_Computing_SDK$/C. After all programs are successfully compiled, we can run them in the directory $NVIDIA_GPU_Computing_SDK$/C/bin/linux/release.

Note that the CUTIL utility is used within the CUDA examples with the inclusion of the matrixMul example. There are helpful and platform independent functions such as cutilSafeCall or cutilCheckError. These functions are described in table 2.

We should keep in mind that in order to create a new CUDA project, the prepared Makefile from $NVIDIA_GPU_Computing_SDK$/C/common should be copied to your project directory. Also we can use the platform independent CUTIL function described above.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutilSafeCall</td>
<td>The macro cutilSafeCall checks if a CUDA API function returns an error or successful state.</td>
</tr>
<tr>
<td>cutilCheckError</td>
<td>The macro cutilCheckError prints error message to stderr output. There are three parameters err, file and line in the macro.</td>
</tr>
</tbody>
</table>
cutilSafeThreadSync  
The macro calls a synchronisation method  
for synchronize “host” threads and checks  
the result. If an error occurs then prints error  
message to stderr output.

cutilSafeMalloc  
The macro cutilSafeMalloc checks a pointer  
to allocated host memory. If the pointer is  
null the method cutilSafeMalloc prints error  
to stderr output.

cutGetMaxGflopsDeviceId  
The function cutGetMaxGflopsDeviceId returns device id (“device” means GPU device  
capable CUDA technology) that has best of  
Gflops performance. It is very helpful (and  
common use) function.

#include "cutil_inline.h"
...
void main(int argc, char** argv) {
...
  cudaSetDevice(cutGetMaxGflopsDeviceId());
...
  float* d_A;
  cutilSafeCall (cudaMalloc((void**) &d_A, mem_size_A));
  float* d_B;
  cutilSafeCall (cudaMalloc((void**) &d_B, mem_size_B));
...
  cutilSafeCall (cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice));
  cutilSafeCall (cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice));
...
  float* d_C;
  cutilSafeCall (cudaMalloc((void**) &d_C, mem_size_C));
...
  matrixMul<<< grid, threads >>>(d_C, d_A, d_B, uiWA, uiWB);
...
  cutilSafeCall (cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost));
...
  cutilSafeCall (cudaFree(d_A));
  cutilSafeCall (cudaFree(d_B));
  cutilSafeCall (cudaFree(d_C));
}

Listing 5: simple CUDA application

3.4 Compilation and the binary output analysis

It is important to know how to compile CUDA applications. The key tool for compilation is the  
derivate of C compilor called nvcc. There are other tools such as cuda-gdb or Visual Profiler that  
are described below in Section 3.5 on page 16. Listing 15 provides a simple code compilation of a  
cuda program. The nvcc produced a binary file
cuda_program that can be executed from the command line. Note that when a cuda application is running, the target GPU may not be used within the X server. Otherwise, the cuda application will not work properly.

The nvcc can generate more than an executable application such as PTX or CUBIN file that are very helpful for development of a cuda application. These files can be utilized to analyse or to make optimizations to the application respectively. Figure 15 describes compilation schema of the nvcc compiler. The nvcc compiler basically generates two types of code. One of them is a CPU code that is normally compiled into the executable application. The other one is a code for GPU devices. The code for the GPU called PTX (Parallel Thread Execution) depends on the kernel's source code in the application. The PTX code is like an intermediary code that has to be translated into a specified target code called a CUBIN. The CUBIN code is linked with the specified CUDA architecture. In other words the CUBIN code that is generated from the PTX code will be different for 2.0 compute capability in comparison to CUBIN dedicted to 1.0 compute capability. It implies that a programmer thinks about the target GPU through the compilation process and has to know which compute capability is used on the GPU. Of course, the nvcc provides other parameters described in Table 3.

Listing 6: nvcc compiler

![nvcc compiler diagram](image)

Figure 3: the nvcc compiler schema

Now we will generate the PTX code as we can see in listing 7. The next listing 8 shows the generated PTX code from the matrixMul project. Keep in mind that there is a short version of the matrixMul PTX code. The PTX code is very similar to assembler-like languages. There are many similar instructions within mul, add, sub and others.
These instructions must be adapted to the specified GPU within the specified CUDA architecture (i.e. sm_13, sm_20). Now we can generate a cubin file from ptx as in Listing 9 that contains a binary code for the specified GPU device.

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA compilation to C source file</td>
<td>-cuda</td>
<td>&quot;c&quot; appended to source file name, as in x.cu.c</td>
</tr>
<tr>
<td>C/C++ compilation to object file</td>
<td>-c</td>
<td>Source file name with suffix replaced by &quot;o&quot; on Linux, or &quot;obj&quot; on Windows</td>
</tr>
<tr>
<td>Cubin generation from CUDA source files</td>
<td>-cubin</td>
<td>Source file name with suffix replaced by &quot;cubin&quot;</td>
</tr>
<tr>
<td>Ptx generation from CUDA source files</td>
<td>-ptx</td>
<td>Source file name with suffix replaced by &quot;ptx&quot;</td>
</tr>
</tbody>
</table>

Table 3: The nvcc parameters

nvcc -ptx matrixMul_kernel.cu

Listing 7: The PTX code generated from the kernel source code

```
...  
cvt.s32.u16  %r1, %ctaid.x;
mul24.lo.s32  %r2, %r1, 16;
cvt.s32.u16  %r3, %ctaid.y;
ld.param.s32  %r4, [__cudaParm__Z9matrixMulPfs_S_ii_wA];
mul.lo.s32  %r5, %r3, %r4;
mul.lo.s32  %r6, %r5, 16;
add.s32  %r7, %r6, %r4;
sub.s32  %r8, %r7, 1;
cvt.s32.u16  %r9, %tid.x;
...  
```

Listing 8: The PTX code example

nvcc -cubin matrixMul_kernel.cu

Listing 9: The CUBIN code generated from the PTX code

3.5 Debugging and Compute Visual Profiler

Debugging is a process of finding bugs and checking if a program is behaving as we expected. Therefore the CUDA Toolkit is supported by its own debugger called cuda-gdb. The cuda-gdb is based on the debugger GNU gdb [3]. It implies that we use the same commands and shortcuts like as in GNU gdb. There are a few specific commands that are not include in the GNU gdb. These commands will be discussed in the next paragraph. Furthermore, there is also a Compute Visual Profiler. The Compute Visual Profiler is dedicated to the optimization of our program. For example, we can use the
Compute Visual Profiler to discover an uncoalesced memory access and then refactor a code responsible for the uncoalesced memory access.

First of all, we need to compile our program with debugging symbols if we want to use the cuda-gdb. We will try to debug the matrixMul project. The cuda-gdb starts in the same way as gdb. In listing 10 we can see a complete usage of the cuda-gdb including debug compilation. There are several new commands, i.e. cuda family commands, info family commands and a thread. If we want to change a thread coordinate within a kernel we should type cuda thread(x,y,z). Another example is to change the device or warp using the command cuda device nd warp nw where nd is the number of the device and nw is the number of the warp. A very helpful command is info. We can simply use info cuda kernel to get some basic information about the running kernel. There are few other words like system, device or sm we can use in the command info cuda.

```
$ cd $NVIDIA_GPU_Computing_SDK/C
$ make gdb=T
cuda-gdb matrixMul
...
(cuda-gdb) break matrixMul
Breakpoint 5 at 0x403c27: file ./matrixMul_kernel.cu, line 41.
(cuda-gdb) run
(cuda-gdb) cuda thread (15,0,0)
(cuda-gdb) cuda device 0 sm 1 warp 2
(cuda-gdb) info cuda kernel
```

Listing 10: Introduction to the cuda-gdb

The Compute Visual Profiler is focused on the profiling and performance of a CUDA application. In other words the Compute Visual Profiler explores a CUDA application and helps us find the places in the application that are low in time (or memory) efficiency. The Compute Visual Profiler helps us increase the CUDA application performance. For example, our CUDA application has uncoalesced memory access in a kernel. We have detected the uncoalesced memory access after analyzing the CUDA application through the use of the Compute Visual Profiler and changed the uncoalesced memory access into a coalesced memory access (described in the section 2.2.5 Memory and coalesced access on page 8). There are several reports like a Profiler Counter Plot, a GPU Time I Height Plot, a GPU Time Width Plot and a GPU Time Summary Plot in the Visual Compute Profiler. This tool has a graphical user interface (GUI) and we don’t use it in the command line. The Compute Visual Profiler requires QT toolkit version 4.5.2 on the unix like system.
4 Application area of CUDA

CUDA technology provides massively parallel compute solution for research or industrial usage. The CUDA based system is able to help increase performance in many branches of industry such as medical, finance, video and photos or new energy. For example, there are fast recalculating options for the financial industry or dramatically improved chances of successful drilling for new energy exploration. Also CUDA technology helps to dramatically accelerate virus simulation.
5 JCUDA wrapper

JCUDA offers binding to CUDA technology on java platform. In other words the JCUDA library is a wrapper for the CUDA library. It supports two different programming interfaces (described in section 2.2.1 The CUDA Architecture overview on page 5). Therefore, a java programmer can use both CUDA Runtime API and CUDA Driver API. The latest version of supported NVIDIA Compute SDK is 3.2.

5.1 Installation

The installation of JCUDA library consists of 3 steps:

- Install required libraries and tools such as jdk, gcc, make or cmake
- Download the file JCUDA-All-0.3.2a-src.zip from a website http://www.jcuda.org/
- Extract the archive JCUDA-All-0.3.2a-src.zip to accessible directory.
- Compile the JCUDA library from a source code.

These steps described above are adapted to a debian-like system we can see on Listing 12. The most complicated step for us is the last step (the compilation of the JCUDA library). Unfortunately, the JCUDA library did not find the path for the variables CUDA_CUDPP_INCLUDE_DIR and CUDA_CUDPP_LIBRARY. Instead of a correct set of these variables, the cmake printed a similar error report to Listing 11. Therefore we have to set these variables manually and continue to compile the JCUDA library (again run cmake).

...  
CMake Error: The following variables are used in this project, but they are set to NOTFOUND. Please set them or make sure they are set and tested correctly in the CMake files:  
CUDA_CUDPP_INCLUDE_DIR (ADVANCED)  
used as include directory in directory /home/lukas/work/JCUDA/JCUDA-All-0.3.2a-src/JCuJ
CUDA_CUDPP_LIBRARY (ADVANCED)  
linked by target "JCUDA-All-1.x86_64" in directory /home/lukas/work/JCUDA/JCUDA-All-0.3.2a-src/JCuJ

...

Listing 11: The cmake error report

```
$ sudo apt-get install build-essential linux-headers-sun-java6-jdk sun-java6-jre cmake
$ wget http://www.jcuda.org/downloads/JCUDA-All-0.3.2a-src.zip
$ unzip JCUDA-All-0.3.2a-src.zip
$ cd JCUDA-All-0.3.2a-src
$ cmake .
$ sed -e 's/CUDA\_CUDPP\_INCLUDE\_DIR\=/NOTFOUND/;s/\$\(home\/lukas\/NVIDIA\/_GPU\_\_Computing\_SDK\/_V\_common\/_incl\_cudpp\/_g\)/' 
```
5.2 Development with JCuda

After successful installation of the JCuda library as mentioned above, a JCuda sample application is described in this section. The JCuda sample application demonstrates using JCuda binding to CUDA Runtime API within the loading of a CUBIN file. Concretely the JCuda sample application has loaded a CUBIN file containing a compiled kernel (depending on the version of compute capability) and passes as kernel arguments a generated 2D float array and its size. As a result of the kernel, the sum is one-dimension of the float array in each thread of the kernel (each element of the 2D array is pointed to another 1D array) and each sum is saved to a result array. The result array is returned to the main program (java program) and it is checked. The most important code fragments are in Listing 13. In other words the JCuda sample application fulfills several tasks:

- Checks if exists the CUBIN file. If does not exist the application has compiled the CUBIN file from the kernel source code (postfix .cu).
- Generates the input 2D array and allocate both host and device memory spaces dedicated to input and output 2D arrays.
- Calls the kernel and pass the arguments such as the input 2D array and its size.
- Returns the output (result) array
- Checks the result array
- Frees allocated memory both host and device memory spaces.

We can see the kernel of the JCuda sample application in Listing 14.

```java
... File cubinFile = new File(cubinFileName);
...
int numThreads = 8;
int size = 128;
...
CUdeviceptr hostDevicePointers[] = new CUdeviceptr[numThreads];
for (int i = 0; i < numThreads; i++)
{
    hostDevicePointers[i] = new CUdeviceptr();
    JCudaDriver.cuMemAlloc(hostDevicePointers[i], size + SizeOf.FLOAT);
}
...
```
for (int i = 0; i < numThreads; i++)
{
    JCudaDriver.cuMemcpyHtoD(hostDevicePointers[i],
                           Pointer.to(hostInput[i]) , size * sizeof.FLOAT);
}

CUdeviceptr deviceInput = new CUdeviceptr();
JCudaDriver.cuMemAlloc(deviceInput, numThreads * sizeof.POINTER);
JCudaDriver.cuMemcpyHtoD(deviceInput, Pointer.to(hostDevicePointers),
                         numThreads * sizeof.POINTER);

JCudaDriver.cuLaunch(function);
JCudaDriver.cuCtxSynchronize();

boolean passed = true;
for (int i = 0; i < numThreads; i++)
{
    float expected = 0;
    for (int j = 0; j < size; j++)
    {
        expected += hostInput[i][j];
    }
    if (Math.abs(hostOutput[i] - expected) > 1e-5)
    {
        passed = false;
        break;
    }
}

for (int i = 0; i < numThreads; i++)
{
    JCudaDriver.cuMemFree(hostDevicePointers[i]);
}
JCudaDriver.cuMemFree(deviceInput);
JCudaDriver.cuMemFree(deviceOutput);

Listing 13: The JCuda sample application (only code fragments)

extern "C"
__global__ void sampleKernel(float* globalInputData, int size, float* globalOutputData)
{
    const unsigned int tidX = threadIdx.x;
    globalOutputData[tidX] = 0;
    for (int i=0; i<size; i++)
    {
        globalOutputData[tidX] += globalInputData[tidX][i];
    }
}

Listing 14: The sample kernel
We have used JCUDA classes and their methods. We can see the most important JCUDA classes and their descriptions in Table 4. The JCUDA documentation is very clear and understandable. There are several JCUDA examples to learn JCUDA usage in the JCUDA site.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JCUDA</td>
<td>The JCUDA offers methods to binding to CUDA Run-time API.</td>
</tr>
<tr>
<td>JCUDA Driver</td>
<td>The JCUDA offers methods to binding to CUDA driver API.</td>
</tr>
<tr>
<td>Pointer</td>
<td>The Pointer is wrapper to C-like pointer to a variable.</td>
</tr>
<tr>
<td>dim3</td>
<td>It use both host memory and device memory spaces.</td>
</tr>
<tr>
<td>Sizeof</td>
<td>The class dim3 wrap C-like datatype dim3 from the CUDA toolkit.</td>
</tr>
<tr>
<td>Package jcuda.utils</td>
<td>Useful class contains size of the variable type such as float, double, short and others.</td>
</tr>
<tr>
<td></td>
<td>Useful set of classes not directly supported by GPU Computing SDK. For example, there is a class KernelLauncher dedicated to manage a kernel function.</td>
</tr>
</tbody>
</table>

Table 4: The JCUDA classes

5.3 Unit testing

The JCUDA application wants to be tested. A java programmer can use a JUnit framework as a unit test framework. The JUnit framework provides base unit test functionality. Moreover, the JUnit framework has very good documentation, so the java programmer adapts to using the JUnit framework quickly. Unfortunately, the JUnit framework does not support a mocked object. The mocked object is a common use technique by a programmer in the unit test. Therefore, they have founded other frameworks like a Powermock framework. There is a huge functionality around unit testing and mocking object in the Powermock framework. Note that the Powermock framework includes the JUnit framework for basic unit testing and EasyMock or Mockito for the mocked objects. In some cases we want to mock JCUDA classes for the unit test coverage. For example, we should cover our JCUDA application by unit tests. There is a situation that we want to know if we call method cudaMemcpyHtoD (which is dedicated to memory copy from host to device memory space) in the unit test. We can see the fragment of the JCUDA application unit test in listing 15. Note that we have to use exactly the same pointer to a variable “a” (means same instance of a class Pointer) both expect method call and run method call. Otherwise the Powermock framework could print java exceptions as the output and our unit test would have failed.

```java
...  
@RunWith(PowerMockRunner.class)
@PrepareForTest(value = {JCUDA.class})
public class KernelTest {
```
... public void simpleKernelTest() {
    Tested tested = new Tested();
    float[] a = new float[2];
    CUdeviceptr ptr = new CUdeviceptr();
    Pointer pointer = Pointer.to(a);
    mockStaticPartial(JCudaDriver.class, "cuMemcpyHtoD",
        CUdeviceptr.class, Pointer.class, Long.TYPE);
    expect(JCudaDriver.cuMemcpyHtoD(ptr, pointer, 8)).
        andReturn(CUresult.CUDA_SUCCESS);
    replayAll();
    tested.run(ptr, pointer);
    verifyAll();
}

Listing 15: Mock JCuda classes
6 Introduction to Multiagent system

In this section the base concept of multiagent simulations (MAS) is described, i.e. what are agents, ACL messages, ontology, performative or how we can develop within JADE framework.

6.1 Basic concept of MAS

In general, the multi agent system provides a solution on how to describe a complex environment composed of many (up to million) independent (possibly intelligent) units. These units are called agents. In other words, the multi agent system is an environment for the agents (the agents are described in section 6.3 Agents below). Each agent interacts with each other. Moreover the agents can interact not only with other agents but with human or other systems. The multi agent system provides a way how the agents communicate, maintain knowledge about the some domain or how the agents can be mobile. Therefore the organization called FIPA (The Foundation for intelligent Physical Agents) was founded. The main goal of the FIPA organization are to define and maintain several standards in order to interoperability between agent-based technology.

6.2 MAS Frameworks

There exists many frameworks which offer functionality to create and run multi-agent systems such as JADE or MASON. These frameworks can maintain many agents (up to thousands) even if both JADE and MASON are written in the java platform. One of the advantages of both of these frameworks is their platform-independence and possibility to write our MAS application in java platform. Otherwise, MASON is not like a JADE framework. MASON is a simulation toolkit for simulating multi agent based environment only. In this paper we focus on the JADE framework because of its fully implemented FIPA standard.

6.3 Agents

As mentioned above the agents are units in a complex environment. The agents have intelligence (from finite state machine up to artificial neural network) and they are autonomous. The agent can communicate with each other. The multi agent simulation library like the JADE handle the communication between agents. The agents may cooperate or be against other agents and a human, respectively. In summary, there are properties that each agent can possess:

- autonomous
- social
- reactive
- proactive
The autonomous means each agent behavior is based on its own decision-making artificial intelligence. The artificial intelligence can be a finite state machine, based on fuzzy logic or artificial neural network (it may include up to thousands of neurons). The agent is interactive with its environment according to collaboration or if it is against another participant. The participant can be another agent and a human respectively. Therefore we say the agent is social. If something has changed in the environment where the agent has lived than the agent reacts to this change. In other words the agent is reactive. The agent not only reacts to its environment but also reacts to other participants of the environment. For example, an agent asks another agent about the services it offers and the other agent should be answered (reacted). We say an agent is proactive if the agent assumes the responsibility for a task and other agents participate in the task. The agent can be mobile also. For example, the agent has moved because of the network node, where the agent was living, is shut down. Moreover the agent has adapted and learned its new goal or changed environment if necessary.

6.4 Ontology

An ontology in the multi agent system means a set of vocabulary that an agent has understood. Consequently, the agent has certainly described the domain of a problem based on this set of vocabulary and shares it with each other. In other words the ontology is a central pivot of the communication act in the multi agent system and gives us a tool how to communicate with agents. The main idea of the ontology is a way how the agents can share knowledge, describe the domain of a problem and exchange tasks, ask or answers each other. The FIPA organization has defined several standards about ontology (such as Ontology Service Specification standard) and other parts of the communication act are described in the next section such as performative or ACL messages. A framework like the JADE implements these standards. The figure 4 describe the basic concept of the ontology defined by the FIPA standards.

A programmer can use basic ontology defined by the FIPA organisation or implement his/her own ontology. Moreover the programmer has implemented his own ontology that is based on the FIPA ontology. In other words ontology can be a hierarchical structure to share the agent knowledge as we can see in figure5. The FIPA ontology standard has defined three basic kinds of ontologies. There are top-level ontologies, domain ontologies and task and application ontologies in the standard. The top-level ontologies contain several very basic general concepts such as space, time, matter, object, event, action. The domain ontologies and task define a vocabulary that describe some specific domain of a problem. For example, the ontology at this level includes domains such as
finance, new energy or medicine and tasks like selling or buying. The application ontologies define the specialization of domains and tasks.

6.5 Performative

A performative is defined as the type of communication act between agents. In other words an agent has to know which type of message is received in order to provide a correct reaction to the received message. There are several basic performatives that FIPA has defined: Confirm, Disconfirm, Failure, Inform, Request, Request When, Request Whenever, Not understood, Query If, Query ref.

Confirm means a sender agent believes that the subject of a communicative act is true where the receiver agent is uncertain about the subject. Disconfirm means a sender agent believes that the subject of communicative act is false where the receiver agent is uncertain about the subject. The subject of the communicative act was not completed by a sending agent. Inform means a sender agent informs that the subject of the communicative act is true. Request means the sender agent wants to perform some action. Request when performative means the sender agent is asking the receiver agent if a subject is true or not. If the agent wants to perform some action whenever a subject is true then a request is sent whenever it is performative. At the end not understood means either the sender agent did not understand received message from other agent or the sender agent informs that it did not understand the performed act (the receiver agent performed act which was not understood to the sender agent).
6.6 ACL Messages

The FIPA organization has defined a message structure for a communicative act. A ACL (Agent Communication Language) message has the structure as in the following list:16. There are several (mandatory) parameters in each ACL message such as sender, receiver, ontology, language, protocol, content, action. The sender and receiver is an agent that send and receive the ACL message respectively. The Parameter ontology has defined ontology for understanding what the ACL message means (described above in section 6.4 Ontology on page 25). The parameter language indicates the name of language used in the ACL message (mostly the ACL Message are using FIPA language FIPA-SL). The protocol parameter defines the interaction protocol. There are other parameters such as conversion-id, reply-with, in-reply-to and reply-by.

```prolog
(request
  :sender (agent--identifier :name alice@mydomain.com)
  :receiver (agent--identifier :name bob@yourdomain.com)
  :ontology travel--assistant
  :language FIPA--SL
  :protocol fipa--request
  :content
  **(action
    (agent--identifier :name bob@yourdomain.com)
    (book--hotel :arrival 15/10/2006
      :departure 05/07/2002 ... )
  )**
)
```

Listing 16: The ACL message structure
7 JADE

As mentioned above the JADE is a java framework that has implemented FIPA standards for rapid development multi agent systems. JADE provides a complete software development kit, it includes robust documentation, samples of code and, of course, an open source library. The JADE framework is under LGPL license which is very flexible for a programmer.

7.1 JADE Architecture

A mutli agent system written in the JADE framework has a platform. The platform participates from containers. In general, a container includes the agents. There is a special container called Main Container. The Main container has two base agents. There is an Agent Management System (AMS) and an Directory Facilitator (DF). We can see the architecture in figure: 6 The JADE architecture. Besides the AMS and the DF the Main Container maintains the Global Agent Descriptor Table (GADT) [7].

The AMS agent is dedicated to agent management (white pages) and the DF (yellow pages) is as a service repository for the agents. In other words the agent services are registered by the DF agent and other agents can find and use these services. Each container holds cache GADT and maintains a Local Agent Descriptor Table (LADT). Note that the agent living in the container can communicate with other agents, moreover these agents can live in other containers. We will see below in the following section 7.3 on page 29 how we can register our agent at AMS agent and register its services at DF agent.

![Figure 6: The JADE architecture](image)

If an agent wants to send an ACL message to another agent, then the container in which the sender agent lives searches first for the local agent description table. If the container does not find the received agent, then it searches the cached global agent description table.
7.2 Installation

In this section we will discuss the installation process of the JADE framework. As you can see in the listing 17, there are two easy steps. First, we download the JADE framework. We need to register at the JADE website and accept their agreement before we start to download. Next we just extract the archive with the JADE framework to a writable directory.

There are several files in the archive. The most important file is jade.jar including JADE classes. Otherwise, the JADE framework requires commons-codec	extsuperscript{2}, therefore the archive also contains this library.

```
download the JADE framework (for example JADE-bin-4.0.1.zip)
$ unzip JADE-bin-4.0.1.zip
```

Listing 17: The installation of the JADE framework

7.3 JADE Agent example

As we can see in the listing 18 there is a sample source code of an agent written in java within the JADE framework. This agent concretely registers our service „Weather-forecast“ in the Yellow pages. Each agent is represented as a java class that extends from a base class Agent. We have overridden a method setup(). The method setup is a bootstrap point of the agent. This method is call within initialization process of the agent. There is registration service in the method setup(). We have used the class DFAgentDescription and ServiceDescription to define our service. Agents that want to use this service need to know the weather-forecast-ontology therefore we have set a service ontology as „weather-forecast-ontology“.

```
public class DFRegisterAgent extends Agent {
    protected void setup() {
        String serviceName = "unknown";
        // Read the name of the service to register as an argument
        Object[] args = getArguments();
        if (args != null && args.length > 0) {
            serviceName = (String) args[0];
        }
        // Register the service
        System.out.println("Agent\"+getLocalName()+\"registering\"+serviceName+\"\"
                +_of_type_\"weather-forecast\"");
        try {
            DFAgentDescription dfd = new DFAgentDescription();
            dfd.setName(getAID());
            ServiceDescription sd = new ServiceDescription();
            sd.setName(serviceName);
            sd.setType("weather-forecast");
            // Agents that want to use this service need to "know" the weather-forecast--
            // ontology
            sd.addOntologies("weather-forecast-ontology");
        }
    }
}
```

\textsuperscript{2}It can be download from http://jakarta.apache.org/commons/codec/.
// Agents that want to use this service need to "speak" the FIPA-\-SL language
sd.addLanguages(FIPANames.ContentLanguage.FIPA_SI);
sd.addProperties(new Property("country", "Italy"));
dfd.addServices(sd);

DFService.register(this, dfd);
}
catch (FIPAException fe) {
    fe.printStackTrace();
}
}

Listing 18: The JADE Agent example

In the listing 19 we see a few steps how our example is compiled of agent and running JADE platform.

$ javac –cp $PATH_TO_JADE/jade.jar:$PATH_TO_JADE/commons–codec/commons–codec–1.3.
    jar DFRегист Agent.java
    jar jade.Boot
    –gui –agents dfrегистAgentDFRegisterAgent

Listing 19: Compilation the agent and run the JADE platform

7.4 JADE Ontology

The ontology represents a java class. There are several base classes such as a Predicate, Concept, Primitive, Aggregate or AgentAction that can extend our ontology class in the JADE framework. In other words each of our ontology classes must be extended from a base ontology class that we can see in figure 7.

![Figure 7: The JADE ontology classes](image-url)
The table 5 give us more in details about the jade ontology classes. For example, we want to define an ontology for a music shop domain. A music shop domain contains CDs, Tracks, Owns and information about sell. A listing 20 shows more details how can we define this ontology.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td>The predicates are expression that can be true or false and describe the status of the world.</td>
</tr>
<tr>
<td>Term</td>
<td>The Term is a generic entity that agents can talk.</td>
</tr>
<tr>
<td>Concept</td>
<td>Concepts are entity that it has complex structure.</td>
</tr>
<tr>
<td>Agent action</td>
<td>The Agent action is a action that can be performed by agents.</td>
</tr>
<tr>
<td>Primitive</td>
<td>The Primitive contains a primitive value such as integer, float, long, short, boolean or byte.</td>
</tr>
<tr>
<td>Aggregate</td>
<td>The Aggregate holds several elements together (similar to list).</td>
</tr>
<tr>
<td>Variable</td>
<td>The Variable represents a variable expression that it is not known yet.</td>
</tr>
</tbody>
</table>

Table 5: The jade ontology overview

```java
// Private constructor
private MusicShopOntology() {
    // The music shop ontology extends the basic ontology
    super(ONT aloneY_ NAME, BasicOntology getInstance());

    try {
        add(new ConceptSchema(ITEM), Item.class);
        add(new ConceptSchema(CD), CD.class);
        add(new ConceptSchema(TRACK), Track.class);
        add(new ConceptSchema(BOOK), Book.class);
        add(new PredicateSchema(OWNS), Owns.class);
        add(new AgentActionSchema(SELL), Sell.class);
        // Structure of the schema for the Item concept
        ConceptSchema cs = (ConceptSchema) getSchema(ITEM);
        cs.add(ITEM_SERIAL, (PrimitiveSchema) getSchema(BasicOntology.INTEGER),
               ObjectSchema.OPTIONAL);
        // The serial–number slot is optional and
        // allowed values are integers.
        // Structure of the schema for the CD concept
        cs = (ConceptSchema) getSchema(CD);
        cs.addSuperSchema((ConceptSchema) getSchema(ITEM));
        cs.add(CD_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING));
        cs.add(CD_TRACKS, (ConceptSchema) getSchema(TRACK), 1, ObjectSchema.UNLIMITED);
        // The tracks slot has cardinality > 1
        // Structure of the schema for the Track concept
        cs = (ConceptSchema) getSchema(TRACK);
        cs.add(TRACK_TITLE, (PrimitiveSchema) getSchema(BasicOntology.STRING));
```
cs.add(TRACK_DUR, (PrimitiveSchema) getSchema(BasicOntology.INTEGER),
        ObjectSchema.OPTIONAL);

    // Structure of the schema for the Book concept
    cs = (ConceptSchema) getSchema(BOOK);
    cs.addSuperSchema((ConceptSchema) getSchema(ITEM));
    cs.add(BOOK_TITLE, (PrimitiveSchema) getSchema(BasicOntology.STRING));

    // Structure of the schema for the Owns predicate
    PredicateSchema ps = (PredicateSchema) getSchema(OWNS);
    ps.add(OWNS_OWNER, (ConceptSchema) getSchema(BasicOntology.AID));
    ps.add(OWNS_ITEM, (ConceptSchema) getSchema(ITEM));

    // Structure of the schema for the Sell agent action
    AgentActionSchema as = (AgentActionSchema) getSchema(SELL);
    as.add(SELL_ITEM, (ConceptSchema) getSchema(ITEM));
    as.add(SELL_BUYER, (ConceptSchema) getSchema(BasicOntology.AID));
}

} catch (OntologyException oe) {
    oe.printStackTrace(); // explains in more detail about the jade behaviours. An example of
    // using behaviours we can see in a listing
}

Listing 20: The music shop ontology

Each agent has a behavior. Moreover it can contain more than one behaviour. There
are several base behaviours such as OneShotBehaviour or CyclicBehaviour in the JADE
framework. Our behaviours must extend from these behaviours. The Table 6 explains in
more detail the jade behaviours. We can see an example of using behaviours in listing 21.

public class SimpleAgent extends Agent {
    protected void setup() {
        System.out.println("Agent_"+getLocalName()+"_started.");
        // Add the CyclicBehaviour
        addBehaviour(new CyclicBehaviour(this) {
            public void action() {
                System.out.println("Cycling");
            }
        });
        // Add the generic behaviour
        addBehaviour(new FourStepBehaviour());
    }
    /**
     * Inner class FourStepBehaviour
     */
    private class FourStepBehaviour extends Behaviour {
        private int step = 1;
        public void action() {
            switch (step) {
                case 1:
                // Perform operation 1: print out a message
                System.out.println("Operation_1");
                break;
                case 2:
                // Perform operation 2: Add a OneShotBehaviour
                System.out.println("Operation_2, Adding one-shot behaviour");
                myAgent.addBehaviour(new OneShotBehaviour(myAgent) {
public void action() {
    System.out.println("One-shot");
}
break;
case 3:
    // Perform operation 3: print out a message
    System.out.println("Operation_3");
    break;
case 4:
    // Perform operation 3: print out a message
    System.out.println("Operation_4");
    break;
}
step++;
}
public boolean done() {
    return step == 5;
}
public int onEnd() {
    myAgent.doDelete();
    return super.onEnd();
} // END of inner class FourStepBehaviour

Listing 21: The example of using behaviours

Table 6: The jade behaviours overview

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompositeBehaviour</td>
<td>An abstract superclass for behaviours composed by many parts. This class holds inside a number of children behaviours.</td>
</tr>
<tr>
<td>LoaderBehaviour</td>
<td>This behaviour serves behaviour-loading requests according to the Behaviour-loading ontology.</td>
</tr>
<tr>
<td>SimpleBehaviour</td>
<td>An atomic behaviour. This abstract class models behaviours that are made by a single, monolithic task and cannot be interrupted.</td>
</tr>
<tr>
<td>WrapperBehaviour</td>
<td>This behaviour allows modifying on the fly the way an existing behaviour object works. The following piece of code provides an example where we modify the done() method of an existing behaviour object to print on the standard output a proper message when the behaviour is completing.</td>
</tr>
<tr>
<td>Behaviour</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OneShotBehaviour</td>
<td>This behaviour process a code only ones. For example, if we want to initialize several parts of the agent we can use this behaviour.</td>
</tr>
<tr>
<td>TickerBehaviour</td>
<td>The TickerBehaviour is useful for periodically executes a certain code. The programmer override method onTick and includes his own code.</td>
</tr>
<tr>
<td>CyclicBehaviour</td>
<td>The CyclicBehaviour is using for process forever (therefore “cyclic”). It can be helpful when we want to check receiving messages or agent states for example.</td>
</tr>
<tr>
<td>WakerBehaviour</td>
<td>The WakerBehaviour is a behaviour similar to OneShotBehaviour. It means that the WakerBehaviour is executed only ones. The main different between the WakerBehaviour and OneShotBehaviour is that the WakerBehaviour is executed after timeout elapsed (“wake up”). Elapsed time has set within a constructor like a Date class or long format.</td>
</tr>
</tbody>
</table>
8 Integration of CUDA kernel code transfer using ACL messages and running of this code

In this chapter we will cover the following goals of the Master's thesis:

- The utilization of CUDA within MAS.
- A CudaAgent Implementation.
- The CudaAgent kernel compilation and data transfer.

8.1 The utilization of CUDA within MAS

We have described CUDA technology and multi agent system in the last chapters. The main advantage of CUDA technology is the power of the graphics processing unit. There are huge numbers of cores that we can use for parallel computing in the GPU. Therefore a task compute on the GPU can be up to 100x faster then the same task compute on the CPU. There exists a number of tasks that we may process on the GPU rather than CPU such as fourier transform and convolution, matrix multiplication, neural network or data mining with large data sets (up to GBs) in the multi agent system.

8.2 Introduction to a CudaAgent

A CudaAgent is the agent that handles kernel source code or PTX (Parallel Thread Execution) code and run on the GPU. The CudaAgent has its own ontology (described below in section 8.4) and it has its own behaviors implemented. In other words the CudaAgent offers to another agent its own service to compile and run the CUDA kernel. The CUDA kernel must be in C source code or PTX code.

As we can see in a figure 8 the CudaAgent after acceptance of the CUDA kernel code (if the code of the cuda kernel is a C source code then the CudaAgent tries to compile it). If the cuda kernel is added and possible compilation of the cuda kernel source code was successful then the addition of the compiled code to its own kernel queue depends on a priority. If the cuda kernel is obtained from the kernel queue than the CudaAgent tries to run the cuda kernel. After the kernel process is finished the CudaAgent sends the ACL message as a result of the cuda kernel. The result can contain data or message information about the kernel failed process. Note that a part of the cuda kernel transfer are also kernel parameters. Each one includes the type of parameter and its data. The parameter should be a primitive type such as float, integer, double and/or other. Moreover the parameter can be an array of these types.

The sender agent expects an ACL message that it contains the result of the cuda kernel such as float array. Otherwise the sender agent could receive an ACL message informing that the sender agent about kernel process failed such as cuda kernel transformation failure, compilation failure or cuda kernel run failure. Each kernel is sent to the CudaAgent through an ACL Message. The kernel code (including input parameters) is converted to
bytes before being sent. The kernel code is converted back when it is received by the CudaAgent.

![Diagram of Cuda kernel transfer]

**Figure 8: The cuda kernel transfer**

### 8.3 The CudaAgent Implementation

The goal of this section is to describe the CudaAgent in more detail. We divide the CudaAgent implementation into three main parts:

- ACL handle messages.
- Kernel managing.
- CudaAgent behaviours.

The part ACL handle messages is implemented as a java class MessageParser and MessageFactory. There is java class Kernel, KernelParameter and KernelBase as part of Kernel managing. CudaAgent behavior is defined as a java class ReceiveMessage and KernelCallManager. There are other classes describe in this Table 7.

### 8.4 The CudaAgent ontology and performatives

The important part of CudaAgent is its own ontology and which performatives it is used. There are several ontology schemas: KernelParameter, KernelBase, KernelCallAction, KernelRegisterAction, KernelUnregisterAction, CudaAgentDescription and CudaResult.
<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CopyDirection</td>
<td>Define if a parameter is input or output.</td>
</tr>
<tr>
<td>CudaAgent</td>
<td>The main class of the CudaAgent project.</td>
</tr>
<tr>
<td>CudaAgentOntology</td>
<td>CudaAgent ontology includes schemas for a class KernelParameter, a KernelBase and kernel actions.</td>
</tr>
<tr>
<td>CudaAgentVocabulary</td>
<td>Interface as vocabulary list used in the CudaAgent ontology.</td>
</tr>
<tr>
<td>CudaResult</td>
<td>CudaResult has used as a answer.</td>
</tr>
<tr>
<td>CudaResultParser</td>
<td>Helper able to parsing result.</td>
</tr>
<tr>
<td>dim3w</td>
<td>Wrapper for a JCUDA class dim3 because of the class dim3 is not serializable.</td>
</tr>
<tr>
<td>GetCudaAgentDescription</td>
<td>An action for short description about the CudaAgent such as number of GPU, performance and other.</td>
</tr>
<tr>
<td>KernelBase</td>
<td>A concept for the CUDA kernel.</td>
</tr>
<tr>
<td>KernelCallAction</td>
<td>An action dedicates to running a CUDA kernel.</td>
</tr>
<tr>
<td>KernelCallManager</td>
<td>A behaviour class dedicates to maintain a CUDA kernel (running kernel, copying in/out data).</td>
</tr>
<tr>
<td>Kernel</td>
<td>Helper for easier manipulating a CUDA kernel. This class extends the class KernelBase.</td>
</tr>
<tr>
<td>KernelParameter</td>
<td>A concept for the kernel parameter.</td>
</tr>
<tr>
<td>KernelPriorityComparator</td>
<td>The class implements a Comparator interface because of kernel comparison.</td>
</tr>
<tr>
<td>KernelRegisterAction</td>
<td>An action dedicates to register kernel by the CudaAgent.</td>
</tr>
<tr>
<td>KernelUnregisterAction</td>
<td>An action dedicates to unregister kernel by the CudaAgent.</td>
</tr>
<tr>
<td>MessageFactory</td>
<td>The class implements Factory pattern to create ACL message such as reply.</td>
</tr>
<tr>
<td>MessageParser</td>
<td>The MessageParser is able to parsing ACL message specific for the CudaAgent.</td>
</tr>
<tr>
<td>ReceiveMessage</td>
<td>A behaviour class dedicates to receive ACL message.</td>
</tr>
</tbody>
</table>

Table 7: The CudaAgent classes

An agent needs to register a cuda kernel by the CudaAgent when the agent wants to process a cuda kernel. There are two ways how the agent can register the cuda kernel. First the agent sends an ACL message as a kernel register action (instance of a class KernelRegisterAction). Next the agent will be informed about successful kernel registration and will also get a kernel ID in this ACL message. After receiving the kernel ID the agent can send a request with the kernel ID for the kernel process as a kernel call action (instance of a class KernelCallAction). The second way is to directly send an ACL message as a kernel call action which contains the whole kernel code with the inclusion of kernel parameters.

The CudaAgent supports other actions such as KernelUnregisterAction and GetCudaAgentDescription. The KernelUnregisterAction is dedicated to unregister cuda kernel. It has to contain a kernel ID. This is a way how we can free memory on the CudaAgent. If the agent wants to know basic information about the CudaAgent (such as number of
devices or its GFlop/s) then the agent sends to the CudaAgent the ACL message as the GetCudaAgentDescription.

The CudaAgent supports these performatives: request, inform, unknown, not understood, agree and failure. Each agent must understand these performatives if it wants to communicate with the CudaAgent.

8.5 The CudaAgent methods and data transfer

The core of the CudaAgent composed from compileKernel, prepareKernel, runKernel and finalizeKernel methods. These methods are the most important methods in the whole CudaAgent. The following Table8 describes important methods of the CudaAgent. There are also implemented behaviors like KernelCallManager and ReceiveMessage. These behaviours are responsible for malignance of the kernel (Listing 22 shows the method responsible for call cuda kernel on the GPU) and receiving ACL messages respectively.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepareKernel</td>
<td>The method initialize kernel. Copy necessary input parameters on the global memory space of GPU.</td>
</tr>
<tr>
<td>runKernel</td>
<td>Run a cuda kernel.</td>
</tr>
<tr>
<td>finalizeKernel</td>
<td>Free allocated memory on the GPU.</td>
</tr>
<tr>
<td>prepareCudaAgentInfo</td>
<td>Make set of basic information about the CudaAgent, mainly about GPU devices.</td>
</tr>
<tr>
<td>setup</td>
<td>Bootstrap method of the CudaAgent.</td>
</tr>
<tr>
<td>getMessageParserInstance</td>
<td>The method return instance of MessageParser with override handle-like methods for manage the ACL messages.</td>
</tr>
</tbody>
</table>

Table 8: The CudaAgent important methods

```java
public class KernelCallManager extends CyclicBehaviour {

    ... protected CudaResult kernelCall(Kernel kernel) throws Exception {
        CudaResult result = null;
        KernelLauncher kernelLauncher = null;

        kernelLauncher = kernel.getKernelLauncher();

        if (kernelLauncher == null) {
            kernel.init();
            cudaAgent.prepareKernel(kernel);
        }

        if (kernelLauncher != null && lastKernelId != kernel.getKernelId()) {
            kernel.copyToDevice();
        }

        cudaAgent.runKernel(kernel);
        lastKernelId = kernel.getKernelId();
    }

    ...}
```
Listing 22: The `kernelCall` method

8.6 Usage of the CudaAgent

We demonstrate usage of the CudaAgent with call `cuda kernel` responsible for matrix multiplication. We have implemented another agent called CudaAgentSample. The CudaAgentSample sends the kernel call action several times and checks result. Before sending the kernel call action the CudaAgentSample must register `cuda kernel` and receive a kernel ID. After knowing the kernel ID the CudaAgentSample can send the ACL message as the kernel call action. In the following list 23 we can see several important code fragments of the CudaAgentSample.

```
public class CudaAgentSample extends Agent {
    ...
    private class HandleResult extends CyclicBehaviour {
        private MessageParser parser;
        private int counter;
        public HandleResult(Agent agent) {
            super(agent);
            counter = 0;
            parser = ((CudaAgentSample)agent).getMessageParserInstance();
        }
        public void action() {
            String exceptionMessage = null;
            ACLMessage reply = null;
            try {
                parser.parseACLMessage();
            } catch (Exception ex) {
                exceptionMessage = ex.getMessage();
                System.out.println("Exception: "+ exceptionMessage);
                parser.makeErrorReply(exceptionMessage);
            } finally {
                parser.sendReply();
            }
        }
    }
}
```

Listing 23: The important code fragments of the CudaAgentSample

8.7 MAS Example

In our example we have run two agents: the CudaAgent and CudaAgentSample. The primary goal of the CudaAgentSample is to transform a cuda kernel code into a CudaAgent, than compile it and run it. The kernel code is matrix multiplication adapted to the parallel CUDA environment. We can divide this goal to tree parts:
• First, the CudaAgentSample tries to register the cuda kernel by the CudaAgent.

• Second, the CudaAgentSample sends to the CudaAgent kernel a call action.

• At least the CudaAgentSample will wait for the response as a kernel result from the CudaAgent.

Moreover we can check performance of the CudaAgent against a non-parallel agent called CpuAgent. The CpuAgent executes the same matrix multiplication code but adapted to CPU environment without parallel execution. The CudaAgentSample tries to execute the kernel code (via kernel call action) several times both by the CudaAgent and the CpuAgent. Next we check result and analyze performance.
9 Evaluation of performance tests and summary results

In this chapter, kernel call and performance tests are described in more detail. These include CUDA kernel call, java emulation kernel call and also summary results of the performance test.

9.1 CUDA Kernel call

We have defined the agent called CudaAgentSample. This agent sends an ACL Message as the cuda kernel call action to the CudaAgent. It contains kernel (source code and ptx code respectively) with the kernel parameters. Listing 24 describes how a kernel call action is sent. We have used the instance of the MessageParser where we have overridden methods that handle the ACL messages. Also we used the instance of MessageFactory for help in creating a message reply. In our example, after sending the cuda kernel call action we have to wait until receiving the reply from the CudaAgent with a kernel id. After receiving the kernel id we can call the kernel within different parameters again using the kernel id. But we do not send all the contents of the kernel because the kernel id exactly identifies the kernel on the CudaAgent after receiving the reply of the cuda kernel call action.

The second way how we can send kernel data to the CudaAgent is using the kernel register action (Listing 25). After sending the kernel register action we received a reply (same reply as the reply after sending the kernel call action) containing the kernel id via overridden handle methods of instance of the class MessageParser (described in the next section). This action just sends the kernel with the parameters to the CudaAgent but the kernel will not be run. If we want to run this kernel we send another ACL message as the kernel call action with the kernel id. The difference between the kernel register action and the kernel call action is shown in Table 10.

```
... message = messageParser.createMessage(new KernelCallAction(preparePerformanceKernel()));
System.out.println("The_KernelBaseAction_has\_been\_created");
...
send(message);
System.out.println("The_ACLMessage\_has\_been\_sent");
...
```

Listing 24: Sending the ACL message as the KernelCallAction

```
...
message = messageParser.createMessage(new KernelRegisterAction(preparePerformanceKernel()));
...
send(message);
System.out.println("The_ACLMessage\_has\_been\_sent");
...
//wait until received a reply as the kernel id.
kernelId = receivedKernelRegisterActionReply();
//run the kernel.
```
message = messageParser.createMessage(new KernelCallAction(kernelId));
send(message);
System.out.println("The_ACLMessage_has BEEN_SENT");
...

Listing 25: Sending the ACL message as the KernelRegisterAction

<table>
<thead>
<tr>
<th>Action</th>
<th>Register</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>KernelRegisterAction</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>KernelCallAction</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 10: Different between the kernel register action and the kernel call action

9.2 Kernel result data within the ACL Message

Next Listing 26 shows how to receive a reply as the kernel result data within the instance of MessageParser. Moreover, each reply may be indentified with the conversation id. In other words, the conversation id identifies an ACL message and its reply. The kernel result data represents parameters that were marked as the output (a constant COPY_TO_HOST) parameter in the kernel call action or possibly in the kernel register action. The instance MessageParser handles the reply of the ACL message and extracts its content as the kernel result data. If a transformation fails the CudaAgent sends an ACL message with failure performative.

```
return new MessageParser(this, false) {
    //override MessageParser methods such as handleInform, handleAgree,
    makeNotUnderstoodReply
    protected void handleInform(CudaResult cudaResult) throws Exception {
        //check result type
        if (cudaResult.getType() == CudaResult.RESULT_TYPE_KERNEL_DATA) {
            //parse the instance of the class CudaResult which contains the kernel result data
            ...
        }
    }
};
```

Listing 26: receive the kernel result data

9.3 Java emulation kernel call

We need to implement the emulation kernel call to compare the performance of the CUDA technology with performance of the java platform. CudaAgent supports CUDA technology for the kernel call. We define a new class as another Agent called AgentCPU that extends the class CudaAgent and overrides several methods for kernel call. Instead of kernel call on the CUDA capable device we call the kernel on the java platform. This kernel is adapted to the java platform, which means that the kernel supports only one thread. Moreover the kernel is compiled into java byte code.
9.4 Performance tests and summary results

In this section, a performance test scenario and summary results are described. A central point of the performance test is using a matrix multiplication algorithm. We have generated two matrices. One of them is 1200x161 and the other one is 80x1600. The multiplication result is saved into a third matrix. We have implemented three agents. These are CudaAgent, the CPUAgent and MatrixAgentSample.

We have measured the time spent on a kernel call via CudaAgent and CPUAgent, respectively, and also the summary time of sending a request to run a kernel (as the kernel call action) and transferring the matrix result via MatrixAgentSample. We have sent the kernel call action 2000 times. We can see the measured data on Figures 9 and 10.

![Time spent on the kernel](image)

Figure 9: time spent on the kernel call
Figure 10: time spent on received kernel result
10 Conclusion

The main goal of this Master's thesis is to find a way how we can safely transfer a CUDA kernel (both the source code of the CUDA kernel, and its ptx code) in the MAS environment from one agent to another specialised agent (called CudaAgent) focused on CUDA technology and process a CUDA kernel on this specialised agent. We have tested the benefits of this solution thanks to the significant safe time on the CUDA kernel process with used parallel computing that could help especially in data mining sub-tasks and other scientific disciplines where we can use parallel computing.

As we expected, the CudaAgent spent less time on the kernel process than the CPUAgent. We can see in figure 9 on the page 43 that the kernel process running on the CudaAgent is approximately 46 times faster then the CPUAgent. Moreover the transfer of the kernel result is also faster on the CudaAgent than on the CPUAgent. The kernel result is transferred approximately 4 times faster between the CudaAgent and the MatrixAgentSample.

The reason why the transfer between the CudaAgent and the MatrixAgentSample is only 4 times faster than that between the CPUAgent and the MatrixAgentSample is the slow copy result data from the device memory space to the host memory space. We could also attribute the result data to the device memory as the pinned memory but it turned out the same measured time as in the case without pinned memory.
11 References


