Rapid code iteration in an Integrated Development Environment with GPU just-in-time compilation

ANDRÉ GRÄSMAN
Rapid code iteration in an Integrated Development Environment with GPU just-in-time compilation

Rapid code iteration i en utvecklingsmiljö som använder GPU just-in-time körning

Author: Gräsman André
grasman@kth.se

Supervisor: Djurfeldt Mikael

Examiner: Laure Erwin

KTH
2014-12-01
Master’s Thesis at CSC
Center for Parallel Computers - PDC
Abstract

Rapid code iteration is a term designating short cycles between changes in the source-code of a program and observing results. This thesis describes an investigation about how an integrated development environment (IDE) can be built in order to gain rapid interaction during software development for graphics processing units (GPUs). The survey has been carried out by implementing an IDE, with a user interface, a compiler, and a runtime in order to provide direct feedback as code is typed.

The presented IDE transforms C-like code to PTX-assembler which is JIT-compiled on a NVIDIA graphics card. Compiling and running a computational intense program about 200 lines of C-like code yields a faster response time than in Visual Studio with either CUDA or C++ using SDL-threads. The program performs RSA encryption/decryption on a large image (11.625MiB) by dividing partial data blocks on different cores on the GPU. The faster response time (more rapid code iteration) is achieved by compiling less code of a smaller language, and using a recycled runtime environment between code iterations. The feedback is measured by the time it takes to compile a change in the source code, plus the time it takes to evaluate the computation.

The IDE provides feedback within 150 milliseconds compared to Visual Studio using CUDA which demand 2 400 milliseconds to provide a response for the same change in the source-code. The majority of the speedup is from the compile time which is 2 100 milliseconds within Visual Studio and CUDA, compared to 13 milliseconds within the presented IDE. Comparing run time of the computation yields a speedup of five times compared to a corresponding C++ SDL-threaded CPU implementation. Comparing run time with CUDA yields a tie.
Sammanfattning

Rapid code iteration i en utvecklingsmiljö som använder GPU just-in-time kompilering

*Rapid code iteration* är en benämning för korta cykler mellan en förändring av källkoden för ett program och möjligheten att observera resultatet från en körning av programmet. Detta examensarbete undersöker hur en integrerad utvecklingsmiljö (IDE) kan konstrueras för att uppnå korta cykler vid utveckling av mjukvara som körs på grafikkort. Undersökningen har genomförts genom att implementera ett IDE med ett användargränssnitt, en kompilator och ett runtime för att ge direkt respons samtidigt som kod skrivs.


# Table of Contents

1. Introduction ........................................................................................................................................ 1

1.1 Rapid code iteration with GPU support today .................................................................................. 2

2. Background on native GPU software development ............................................................................. 5

2.1 NVIDIA's parallel processing architecture CUDA ............................................................................. 5

2.1.1 The Kernel as a GPU program ...................................................................................................... 7

2.1.2 Memory configuration .................................................................................................................... 8

2.1.3 PTX Assembler as a GPU program ................................................................................................... 10

3. Survey design ........................................................................................................................................ 11

3.1 Creating an environment to test rapid code iteration ........................................................................ 12

4. Engine implementation description ..................................................................................................... 15

4.1 Parsing ................................................................................................................................................. 15

4.1.1 Tokenization .................................................................................................................................... 15

4.1.2 Tree Building .................................................................................................................................. 17

4.2 Type Checking ..................................................................................................................................... 20

4.2.1 Operands and evaluation of constant expressions ......................................................................... 21

4.2.2 Selection by Reference .................................................................................................................... 22

4.3 Code intelligence generation .............................................................................................................. 23

4.4 Intermediate Representation ............................................................................................................. 24

4.4.1 PTX generation ............................................................................................................................... 24

4.5 Runtime ............................................................................................................................................... 24

5. Results of evaluating the IDE .............................................................................................................. 25

5.1 Drawing a 3D-surface ............................................................................................................................ 25

5.2 Image filtering ..................................................................................................................................... 26

5.3 RSA encryption/decryption .................................................................................................................. 27

5.3.1 RSA implementation details ............................................................................................................ 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Benchmark</td>
<td>30</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Compile time details</td>
<td>31</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Run time details</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Discussion</td>
<td>37</td>
</tr>
<tr>
<td>6.1</td>
<td>Fast response</td>
<td>37</td>
</tr>
<tr>
<td>6.1.1</td>
<td>PTX versus NVCC compiled CUDA binaries</td>
<td>38</td>
</tr>
<tr>
<td>6.2</td>
<td>About the compiler design</td>
<td>38</td>
</tr>
<tr>
<td>6.3</td>
<td>Future development</td>
<td>39</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Infinite kernel loop instability</td>
<td>39</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Implementation of new language features</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>Bibliography</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>Appendix</td>
<td>43</td>
</tr>
<tr>
<td>8.1</td>
<td>Grammar of YouEngineer</td>
<td>43</td>
</tr>
<tr>
<td>8.2</td>
<td>Drawing 3D surface (YouEngineer code)</td>
<td>48</td>
</tr>
<tr>
<td>8.3</td>
<td>RSA encryption/decryption (YouEngineer code)</td>
<td>49</td>
</tr>
<tr>
<td>8.4</td>
<td>RSA encryption kernel (YouEngineer generated PTX)</td>
<td>52</td>
</tr>
<tr>
<td>8.5</td>
<td>RSA decryption kernel (YouEngineer generated PTX)</td>
<td>53</td>
</tr>
<tr>
<td>8.6</td>
<td>RSA encryption/decryption (NVCC generated PTX)</td>
<td>54</td>
</tr>
</tbody>
</table>
1 Introduction

This chapter introduces rapid code iteration and the purpose of achieving this in software development for graphics card.

Being a programmer and having a computer is like being a governor having a butler, with the advantages that there is much less ethical complications to control the computer. One similarity between the governor and the programmer is the difficulty to communicate instructions properly, especially when less about the task is known in advance and details must be thought out along the way. Sometimes the only thing to do is to communicate a hint of an instruction and wait for the response in order to learn how a better instruction can be formulated. In many cases this is how software development is carried into practice. The possibility of being able to designate a direction with code and observe the result very fast in order to do it again is termed rapid code iteration [1].

Rapid code iteration is not important for all projects. It is less important when long planning and large implementations are required between tests or the execution of a program involves human interactions to run. On the other hand it is of importance when the behavior of a single algorithm is explored or a parameter needs to be tweaked right. Such scenarios rise naturally in fields where experimental algorithm development is exercised.

Rapid code iteration is useful because it saves time. It also saves the programmers state of mind which leave more focus to the actual task. How well the process can be achieved depends on the development environment in use as well as the programming language. Environments associated with MATLAB, Python, Java, and C# are good at this. But their support to make use of the graphic processing unit (GPU) on the graphics card as a parallel computing device for algorithm development is none or limited. To achieve rapid code iteration for development of GPU adapted algorithms there must be support in the language to grasp beneficial features of the hardware, as well as an interactive development environment to write and run code from. The importance of the environment cannot be emphasized enough. Writing compilation commands in a terminal compared to pushing a button (or no need to push any button at all) is a larger interruption from the ongoing process.

A personal computer has a central processing unit (CPU) and a GPU which complements each other to manage parallelism on different levels. The CPU has a central role in the computer (as the name implies) and contains a small number of advanced cores with a large set of features. The GPU is placed on a graphics card and has hundreds to thousands of more simplified cores. These cores are able to execute a shared program and operate on individual and shared memory. The architecture makes the GPU suitable for a large number of independent calculations. An example of such calculation is an element wise vector operation (like $f(a_i, b_i) = a_i \cdot b_i$) which is nothing else but an individual program running several times, in any order with different sets of input.

A few years back the GPU development was almost exclusively driven by the game industry which had a demand for cheap parallel hardware. Graphics in games are to a large extent calculated similarly to element wise vector operations where each element is related to a pixel.
value. During the history of graphics card development, GPUs become the target for physics simulations in games as well. The increase in responsibility gained new features which made GPUs suitable for more general parallel computations. Today there are great expectations of outsourcing many kinds of parallel computations to the GPU in order to gain speed and availability to the CPU.

1.1 Rapid code iteration with GPU support today

To achieve rapid code iteration it is of importance to have a short compile time and a user friendly interface to invoke compilation and execution from. It is also important to execute the calculation fast in order to gain a result to view fast. The GPU has proven itself to help with the last part for certain kind of tasks, but development environments have a distance to go before all parts are in place.

GPU executables are largely developed in languages like C and C++ which have a long compile time compared to other commonly used languages as MATLAB, Python, Java, and C#. One of the reasons for the long compile times is the use of header files which is the key to structure projects and describe libraries. C and C++ with its large freedom of expression force headers to be reparsed each time they are included in every file. This must be done since a user defined macro can change the content of the header at any point in the code. This is a good feature in the preprocessor but one of the bottlenecks when it comes to achieving rapid code iteration with C and C++.

MATLAB introduce GPU computing on their website [2] by describing how MATLAB code can be linked to precompiled GPU kernels which are developed in CUDA (a separate environment developed by NVIDIA to build GPU executables with C and C++). MATLABs main approach to benefit from GPU hardware is to present predefined algorithms in their backend library. This is done by allowing the user to declare vectors and matrices on the graphics card to compute operations using the GPU. But MATLAB does not allow the user to design GPU algorithms like a custom made matrix multiplication as an example.

Python supports GPU computations by sending GPU-programs (called kernels) as a string for just-in-time compilation during runtime through an API in an extension library [3]. The process is currently highlighted at gpuscience.com under the section “GPU programming with Python” (http://gpuscience.com/code-examples/gpu-programming-with-python/). But the approach does not allow the language of the kernel to be processed on the same level as the surrounding Python code. The kernel is stored as data instead of instructions which makes the kernels unconnected to the environment and the Python language. The kernels must be processed and compiled during runtime in a later stage compared to the rest of the code. This makes the possibility to support code intelligence as word completion and highlighted contextual keywords in the future complicated.

Java has third party extension libraries similar to Python which makes the GPU accessible without changing or extending the Java language. One of these libraries is JCuda which gives the functionality of the CUDA API when accessing NVIDIA specific graphics card from JAVA. JCuda manages memory resources and allows GPU programs (kernels) to be invoked from
Java. But the kernel itself is not developed in Java; it is developed in CUDA C and compiled using the NVIDIA CUDA C compiler named NVCC [4].

Beside JCuda there are some promising attempts to move the GPU part below the surface in Java. One of these is to detect computation intensive loops within the intermediate assembler and replace it with a GPU invocation [5]. But that solution comes with the problem of finding a decisive threshold of computation intensity at compile time. The decision of using the GPU instead of the CPU may be easy for a programmer to decide but hard for the compiler. The programmer has better knowledge about loop iteration count, computation intensity and commonly used program paths, compared to the compiler. The compiler can gain this knowledge if it is allowed to run the program and profile it, but that will increase the compilation time.

C# has language bindings to access the GPU as well. One of the third party solutions is CUDAfy.NET which allows similar functionalities as JCuda. One difference is the support to run and write GPU kernels directly in C#. CUDAfy.NET makes use of the method attribute in the C# language to provide additional information about methods to the preprocessor and compiler. CUDAfy.NET uses ILSpy to decompile the .NET code and identify methods with a “[Cudafy]” attribute through reflection. Methods with this attribute are considered as GPU kernels, which are translated to CUDA C [6] by CUDAfy.NET. The CUDA C representation of the kernels is finally compiled in an ordinary manner by the NVIDIA CUDA C compiler to assembler that fits the GPU.

Independently of the choice of language or environment, the GPU part is either treated differently from the main approach of development or relies on compilation of C, which is hard to fit under the same roof as rapid code iteration. The main focus in the tools and languages mentioned is to integrate GPU support and yield execution speed, not the best interaction.

The scope of this thesis will focus on the question whether it is possible to build an environment which is able to target the GPU and at the same time provide fast interaction for a simplified C like language. The question is under assumption that focus is on rapid code iteration and all code is parsed and processed from the same language on the same level.

The question raises another question about performance. Namely, is there a reason to target the GPU instead of the CPU minding overheads and minimal time for optimization when the response time is the variable to minimize? How will a rapid compiled GPU program scale compared to an ordinary GPU program compiled with an established toolchain that is known to provide fast execution speed? Where on the timeline does a fully utilized CPU candidate fit in? Is there a reason to target GPUs for rapid code iteration or must the graphics cards get better first?
2 Background on native GPU software development

This chapter gives background on established methods to build GPU applications on a level close to hardware. It describes CUDA and related NVIDIA devices in particular.

There are several methods available in order to develop GPU executables. Some of the most used are DirectCompute, OpenCL and CUDA. DirectCompute is a Microsoft DirectX API which is used to compile shaders in a High-level shader language (HLSL). It will target any hardware supported by DirectX. The language has a rich amount of features which make it possible to use HLSL as a general purpose parallel computing language [7].

OpenCL and CUDA are frameworks using C and C++ instead of HLSL. OpenCL is an open computing language for both CPUs and GPUs while CUDA target NVIDIA specific GPUs only. Since OpenCL is more general they differ in terminologies, otherwise they are closely resembled.

NVIDIA is very market-oriented and has made CUDA easy to start with. It is visible that much effort has been spent in order to achieve simplicity. The software development tool kit (SDK) is easy to install and includes good templates and tutorials which cut down the learning cost significantly.

CUDA's design to reach good performance on one type of hardware is a decisive element to prefer CUDA if the project can accept a lock to NVIDIA hardware. An environment closely coupled with the underlying hardware can take advantage of specific features. In CUDA one of these features is the possibility to run PTX assembler at runtime through the NVIDIA driver API. This feature will be described in detail since it serves as a key component to launch programs on the GPU.

2.1 NVIDIA's parallel processing architecture CUDA

CUDA (Compute Unified Device Architecture) is a general purpose parallel computing platform developed by NVIDIA. It allows C++ applications to be linked together with compiled CUDA code. All CUDA applications launch on the CPU as any traditional application. When computation on the GPU is desired the application creates a link to the graphics card and launches a program. The CPU and the system memory are referred to as the host while the GPU with the memory available on the graphics card is referred to as the device. The host controls the device as well as normal application duties.

In a typical scenario the host allocates two memory arrays on the device (one for input and one for output). The host copies the input data to the first array and launches the CUDA program with pointers to the arrays as parameters. The host waits for the CUDA program to finish and then copies data back from the second array. The time it takes to copy memory back and forth is often the culprit to prefer the CPU before the GPU. But in order to achieve high performance, it is possible to copy data at the same time as the GPU is working on data that already has arrived. This makes it possible to almost hide all communication latency between
the host and device [8] (of larger importance in data stream applications). But this is more of an implementation optimization than a first code praxis.

As mentioned the host is responsible for launching CUDA programs. This is done by calling a function declared as __global__ void. This function is named a kernel and it must be defined in a *.cu file in order to be recognized by the CUDA compiler.

A kernel lives in its own context, which contain predefined members that are accessible from the body of the function. The most commonly used members are the four, three dimensional integer vectors gridDim, blockDim, blockIdx, and threadIdx. Each of them holds X, Y, and Z components that reveal the whole methodology in CUDA. Figure 1 demonstrates a simplified program that shows the syntax of a kernel and how it is launched. The program computes an element wise vector addition.

```c
// Kernel code placed in a *.cu file which run on the device
__global__ void VectorAddition(float* result, const float* a, const float* b)
{
    uint64_t i = blockDim.x * blockIdx.x + threadIdx.x;
    result[i] = a[i] + b[i];
}

// Main C++ program which run on the host
int threadDim = 512;
int blockDim = 16;

// Pointers to global memory on device
float* resultp;
float* ap;
float* bp;

VectorAddition<<< blockDim, threadDim >>>(resultp, ap, bp);
```

Figure 1. Example of a CUDA kernel named VectorAddition and how it is launched.
2.1.1 The Kernel as a GPU program

A kernel is always launched in relation to a mesh of one, two or three dimensions. The kernel can be thought of as a simple function, which will receive a defined number of simultaneous calls. The number of calls will match the number of distinct points, spanned by the point space described by gridDim and blockDim. Each call will execute in its own thread and have its own value on X, Y and Z in the blockIdx, and threadIdx vectors.

In order to allow synchronization, threads are organized in blocks. A block can have one, two or three dimensions as well, and represents a cooperative thread array (CTA). The size of the block is determined by the vector blockDim. Threads in a CTA are able to synchronize the execution with each other by waiting for all members to reach a common instruction. The number of threads within a CTA is limited in hardware. CUDA computability target 1.x must have 512 or less threads per block. But the number was increased to 1024 in version 2.0 (in other words it is possible that this restriction will be ameliorated in the future).

The one, two or three dimensioned mesh is built from blocks. The size of the mesh is determined by the vector gridDim (grid is synonymous with mesh in the CUDA context). On CUDA computability target 1.x and 2.0 the value of X, Y and Z must be less than $2^{16}$. The range of the X component was increased to $2^{31}$ in 3.0. Figure 2 shows an illustration of a two dimensional mesh (called grid) with its cooperative thread arrays.

![Figure 2. Example of a kernel with its grid and cooperative thread array.](URL: http://docs.nvidia.com/cuda/parallel-thread-execution/graphics/thread-batching.png)
2.1.2 Memory configuration
NVIDIA graphics card have memory in different levels with different size, accessibility and speed. The more memory a level contains, the further away it is from the core which make it less accessible (slower).

2.1.2.1 Global memory
The memory accessible from the host is known as the global memory. It is shared among all threads in all blocks. It is the largest but slowest memory on the device. A typical access cost is generally at least the time it takes to execute 100 instructions [9, page 52] and the size is measured in gigabytes.

2.1.2.2 Per-block shared memory
To allow communication with increased speed, there is per-block shared memory which is accessible from all threads within a block, but not between threads across different blocks. The memory has no access cost, but its size is only measured in tens of kilobytes. The per-block shared memory is used together with synchronization instructions to communicate data between threads.

2.1.2.3 Per-thread local memory
The per-thread local memory is not its own physical memory. The address space is mapped to global memory by the compiler [10]. The difference between local and global memory is that local store operations are L1-cached. Because of this the access time of per-thread local memory can be as low as zero, and as high as the access time of global memory, depending on the cache line and cache misses.

L1-cache is a general name of the first cache level that can cause latency to the execution. Both CPUs and GPUs use memory caches on several levels (commonly L1 and L2) to increase performance. Figure 3 illustrates the cache configuration. It shows how several SMs (Single Instruction Multiple Thread, Multiprocessors) has their L1 cache and shared memory close to the core. It also shows how the global memory is accessed through the two cache levels. Figure 4 shows an illustration of the memory hierarchy.

Figure 3. Example of Configuration of a Single Instruction Multiple Thread (SIMT) Multiprocessor (SM).
2.1.2.4 Registers

Data used by a single thread is foremost stored in registers. The NVIDIA Fermi architecture has a limit of 62 usable read/write registers per thread, while the Kepler architecture has 254 [11]. Registers are 32 bit in size which gives each thread access to less than a kilobyte of the fastest memory. If a thread needs more memory than the register space can provide, register spilling occurs. Spilling will offload data to the fast per-block shared memory if space is available. Otherwise the slower per-thread local memory is used.

2.1.2.5 Constant state space memory

In addition to the read/write memory, there is a constant state space, which is read only. The constant state space is shared on the same level as global memory but more restricted in size. The memory is initialized by the host and used to store kernel arguments and other optional parameters. The size depends on the target compute capability but a common configuration is eleven 64KB banks (no alignment between banks) [9, page 28]. The amortized access cost of constant memory is low (0 clocks) but the first access is slower [9, page 52].
2.1.3 PTX Assembler as a GPU program

PTX is a parallel thread execution virtual machine with an instruction set architecture (ISA) designed to be efficient on NVIDIA GPUs [9, page 2]. It is a high-level assembly language with its purpose to provide a stable ISA for multiple GPU generations. It represents the lowest level in order to target NVIDIA devices, and it is built to achieve performance comparable to native GPU performance. NVIDIA have chosen not to reveal individual instruction sets for independent devices. Instead PTX serves as a machine-independent ISA which C++ and other languages can be compiled to. PTX assembler is mapped to specific device instructions by the NVIDIA driver at runtime/compile time.

PTX assembler together with its virtual machine may be compared with just-in-time compilation of shaders in 3D rendering [9, page 52]. In both cases programs can be loaded from source directly, compiled and executed on a GPU during runtime. The difference between PTX and shader languages such as HLSL [12] is the abstraction level and purpose. PTX is designed to be low level and general while HLSL is designed to make it easy to write programs like vertex shaders, geometry shaders, and pixel/fragment shaders.

PTX can be used as inline assembler in CUDA kernels to implement specific optimizations. It allows the programmer to be more specific about the implementation than C++ does. CUDA projects are compiled by NVCC which is NVIDIA's CUDA/C++ compiler. NVCC can either output the whole GPU part as binary Cubin-files, or as PTX assembler [13].

The GPU part does not need to be compiled at the same time as the rest of the CUDA project. The NVIDIA driver contains a virtual machine which is able to just-in-time compile PTX assembler on the fly from a string during runtime. Whole kernels can be written, loaded and executed from PTX this way. The NVIDIA SDK contains an example where this is demonstrated.
3 Survey design

This chapter describes how the investigation of achieving rapid code iteration is carried out by implementing an IDE with a user interface, a compiler and a runtime.

Achieving rapid code iteration requires an investigation of how the proportion of time between compilation, optimization and execution is spent in the best way. The compilation time can be minimized in at least two ways (beside the obvious of optimizing the compiler). One way is to address the language by depriving it on features which take long time to compile. Another way is to reduce the amount of code that needs to be compiled. The second issue can be managed by integrating functions and data types into the compiler. Such action eliminates the need to manage non-project specific code which otherwise come with the use of includes, descriptions of compiled library and non-pre-compiled libraries.

Depriving the language from features is easy to do, but it comes with an expense with regard to usefulness. If the language forbids generics and user defined data types it raises great demands on the types available. But the need for code cuddling and internal book keeping at compile time is extensively reduced. The surrounding environment can be stored statically and allows pre-stored procedures for managing types, methods and members.

The amount of time which can be spent on optimization is extremely large. Some optimization problems are NP-complete or even un-decidable. A practical approach is to run heuristics which may improve the program as long as the programmer is willing to wait. This means that an optimization process rarely produces optimal code even for a generous amount of time [14, page 15-17]. On the other hand, not all optimizations are hard to apply. Pre-evaluation of constant expressions is an example of an optimization which is cheap to perform and yields a good improvement. It is cheap because it is easy to find where it is required, and the evaluation must be done at some point anyway. If it is not done at compile time the expression must be evaluated each time the point of code is visited during run time.

The investigation about rapid code iteration requires an environment which allows all the mentioned parameter dimensions to be adjustable in order to find a good fit. The parameter dimensions between optimization and run time can be addressed (partly) by setting different levels of optimizations in any compiler. But the parameter between usefulness and richness of language features is harder to adjust since it requires modification on the language, which is a major procedure to perform on a compiler. Such a study might easiest be done by constructing a new compiler. Especially if a preset of types, methods and members are to be integrated as well.

A runtime itself can benefit from recycling runtime resources between code iterations. A regular IDE invokes a complete new launch by calling the operating system each time a new program is started for the first time. The runtime which manages the application creates a new process with new allocated memory. All new resources which are used by the new process are reset in order to provide safety and consistent behavior. Memory resources must be zeroed in order to not leak information from a previous process. Some of these reallocations and safety procedures associated with creating a new process are unnecessary compared to what a new version of the same program needs.
3.1 Creating an environment to test rapid code iteration

A simple environment is designed to learn what it can offer *rapid code iteration* by using a basic language, pre-included types and cheap optimization. The implementation does not intend to be full-fledged. It rather intends to constitute a complete chain to make measurements of its performance possible. The implemented environment is a small IDE with a compiler and a runtime. It is named YouEngineer in order to differentiate it from other IDEs which will be mentioned later.

YouEngineer has a user interface which has a code area placed adjacent to a response list that shows results from calculations. The compiler parses code, performs type checking and yields an intermediate representation which is interpreted on the CPU-side and just-in-time compiled on the GPU-side. The compiler generates code intelligence which is used to color contextual keywords in the code area of the user interface and to provide details about the context through an auto complete feature.

YouEngineer tries to provide a direct response simultaneously while code is being typed. This means that every change in the code is passed to the compiler and launched by the runtime if no syntactical errors are found. The typed program runs until it reaches its end or until it is terminated by the IDE to make space for a new execution (caused by a change in code).

The compiled program is stored as an intermediate representation (IR) in two formats. The CPU side uses an interpretable tree structure while the GPU side stores kernels in PTX assembler. The kernel is sent to the NVIDIA driver for JIT compilation and is invoked by the GPU runtime through the NVIDIA driver API.

Figure 5 shows how the compiler, runtime and user interface interact with each other. The compiler on the left side receives code from the interface which is transformed to different representations and yields a response that becomes visible to the user at the right side of the figure.

![Figure 5](image_url)

*Figure 5. Example of how the compiler, runtime and user interface interact with each other.*
The compiler accepts a language which can be regarded as a subset of C that supports declarations of GPU-functions (kernels) in the same manner as CUDA (with blocks and threads). The idea is to yield an environment as similar as possible to maintain the same strategy in problem solving. In order to avoid implementing the whole C-language, YouEngineer have its own tricks to make it useful. To understand details of the language implemented, have a look at appendix 8.1 for a context free language description on Backus-Naur-form.

GPU-functions are distinguished in the code by a "gpu" keyword which allows the function to inherit a different context compared to ordinary functions. The GPU-function context contains the predefined vectors `gridDim`, `blockDim`, `blockIdx`, and `threadIdx` in the same manner as CUDA kernels does to reveal the kernels position within the mesh. The execution is prepared by encapsulating a pointer to the GPU function with dimensions about the desired grid size (blocks and threads) in an object on the CPU side in YouEngineer. The object contains all environmental information needed to start evaluating the function on the GPU.

YouEngineer borrows a feature from MATLAB that makes the mandatory semicolon at the end of each statement in C optional. If the semicolon is removed the statement is printed to the output list during execution. The output will accept types as string, numbers, images and a 3D stage object (used to plot more advanced shapes).

Figure 6 shows an example of the syntax in YouEngineer. The program decelerates a GPU-function that writes an incrementing number to memory. The memory is copied from the device and printed to the output list by the CPU side.
<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gpu void myGpuFunction(bytePtr globalAddress, uint64 myI)</code></td>
<td>4</td>
</tr>
<tr>
<td>{</td>
<td>5</td>
</tr>
<tr>
<td>uint64 myGlobalId = blockSize_x * blockIdx_x + threadIdx_x;</td>
<td>6</td>
</tr>
<tr>
<td>bytePtr myGlobalPtr = globalAddress + myGlobalId * 8;</td>
<td>7</td>
</tr>
<tr>
<td>globalStore_uint64(myGlobalPtr, myGlobalId + myI);</td>
<td>8</td>
</tr>
<tr>
<td>}</td>
<td>9</td>
</tr>
<tr>
<td>myDevice = Gpu.GetMaxGFlopsDevice();</td>
<td>10</td>
</tr>
<tr>
<td>bytePtr deviceMemPtr = myDeviceMalloc(256<em>32</em>8);</td>
<td>11</td>
</tr>
<tr>
<td>// Creates a pointer to GPU-function (no call)</td>
<td>12</td>
</tr>
<tr>
<td>myKernel = myGpuFunction(deviceMemPtr, 4);</td>
<td>13</td>
</tr>
<tr>
<td>// Launch kernel, blockDim(X,Y,Z), gridDim(X,Y,Z)</td>
<td>14</td>
</tr>
<tr>
<td>myDevice.Run(myKernel, 256, 1, 1, 32, 1, 1);</td>
<td>15</td>
</tr>
<tr>
<td>myByteList = myDevice.GetAs_UINT8List(deviceMemPtr, 10 * 8);</td>
<td>16</td>
</tr>
<tr>
<td>for(int i = 0; i &lt; 10; ++i)</td>
<td>17</td>
</tr>
<tr>
<td>{</td>
<td>18</td>
</tr>
<tr>
<td>// This line is printed (no semicolon at the end)</td>
<td>19</td>
</tr>
<tr>
<td>myByteList.ReadAsUINT64(i * 8)</td>
<td>20</td>
</tr>
<tr>
<td>}</td>
<td>21</td>
</tr>
</tbody>
</table>

Figure 6. An example of a small program that demonstrates the language.
4 Engine implementation description

This chapter describes the different components in YouEngineer to yield an interpretable tree structure, and PTX assembler for its runtime.

The compiler and runtime working together is the engine of YouEngineer. The compiler transforms source code from the IDE to an intermediate representation, which is executed by the runtime. The compiler is a pipeline where parsing, type checking and IR building are the main tasks.

4.1 Parsing

Parsing is the process of transforming a string of words to a tree that reveals the syntactical relationship between the words. A common way to build the tree is to make use of grammatical rules and match the string to a language [14, page 60]. YouEngineer uses a different approach, more similar to a pipeline of observations and modifications. The pipeline starts to consider all words as a continuous chain of nodes in a tree without branches. A certain observation will cause a modification that transforms the straight chain to a parsed tree step by step. The process is a multi-pass compilation compared to one-pass compilers that only pass through any part of the code once. One-pass compilers are commonly known to be smaller and faster than multi-pass compilers, which is desirable when aiming at rapid code iteration. But one-pass compilers have disadvantages as well. One disadvantage is the loss of the wide eye that allows better code generation. Another is the impossibility to compile languages that allow declaration of members anywhere in the code (which requires at least two passes). But the decisive factor of using a multi-pass technique instead of a one-pass in this project is the transparency which comes with the design. Dividing operations into several steps with well-defined formats on input and output makes the process easier to grasp.

4.1.1 Tokenization

The first step in order to parse the code is to identify tokens which are the smallest components in a language [14, page 6]. A token is either a word, number or a special character sequence that match a certain word or pattern. Words like if, for, and return are examples of static tokens while identifiers, numbers and string literals are examples of tokens which must be recognized by patterns. YouEngineer use 62 static words and 6 patterns to identify the components of the code to a sequence of classified tokens.

The token sequence is built as a reconstruction of the code by using tokens instead of characters. The static token space is stored as a tree where each node represents one character of a word. A sequence from the root of the tree to a leaf represents one static token, and the depth of the tree is equal to the longest token. Each level in the tree corresponds to a possible observation of a character from the code which leads to at least one match. The levels in the tree are represented as sorted lists to make it possible to perform a dichotomic search in order to distinguish the remaining tokens which can match a growing window of observations.

The tree representation gives a $O(n \cdot \log(s))$ time complexity of matching static tokens where $n$ is the number of characters in the code, and $s$ is the amount of static tokens [15].
The patterns are matched simultaneously as the static tokens are searched. The patterns are represented as a list of finite state machines which is shrinked by removing patterns that no longer match observations. The list of patterns in addition to the static tokens gives a time complexity of $O(n \cdot (\log(s) + p))$ where the new $p$ is the number of patterns. In practice most of the candidates are excluded after the first observation which leaves the rest of the match of a single token cheap to perform.

The search for a suitable token continues as long as there is at least one pattern or static word which can match another observation. In the case when a static token and a pattern match the exact same number of observations, there is a priority lookup table that decides who wins. An example of this is where identifiers matched by patterns always loose to static words as if, for, and return.

Because YouEngineer allows expressions to be printed if they are expressed without an ending semicolon (;), the line break character have a more important role than it usually has in languages. C, C++, C# and Java manage line breaks as any whitespace characters (which only is a delimiter). Line breaks in YouEngineer has to work similar as semicolon in order to state where certain expressions end and therefore line breaks must be managed as a token. But, YouEngineer tokenize ordinary whitespace characters as well. This is an odd behavior but it has its advantages. The preservation keeps all information about the typed program without information loss. It makes it easy to regenerate the code with highlighted errors and colored contextual key words. The reasoning reveals that all types of comments are treated as tokens as well.

Figure 7 shows an example of a token sequence where each line starts with a token type followed by a copy of the code part it represents. The ending @ indicates the starting character position of the code where the token was initialized. The character position is used later to glue contextual meaning into the code.

<table>
<thead>
<tr>
<th>Code</th>
<th>Token Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
<td>- Id s: @0</td>
</tr>
<tr>
<td>a++ // A comment</td>
<td>- Space_T: @1</td>
</tr>
<tr>
<td>a</td>
<td>- Equal_T: @2</td>
</tr>
<tr>
<td>++a</td>
<td>- Space_T: @3</td>
</tr>
<tr>
<td></td>
<td>- Id_Literal_I: @4</td>
</tr>
<tr>
<td></td>
<td>- LineBreak_T: @5</td>
</tr>
<tr>
<td></td>
<td>- Id s: @6</td>
</tr>
<tr>
<td></td>
<td>- PlusPlus_T: @7</td>
</tr>
<tr>
<td></td>
<td>- Space_T: @9</td>
</tr>
<tr>
<td></td>
<td>- LineComment: // A comment: @10</td>
</tr>
<tr>
<td></td>
<td>- LineBreak_T: @22</td>
</tr>
<tr>
<td></td>
<td>- Id a: @23</td>
</tr>
<tr>
<td></td>
<td>- LineBreak_T: @24</td>
</tr>
<tr>
<td></td>
<td>- PlusPlus_T: @25</td>
</tr>
<tr>
<td></td>
<td>- Id s: @27</td>
</tr>
</tbody>
</table>

Figure 7. Example of a token sequence.
4.1.2 Tree Building
The purpose of tree building is to transform the sequence of tokens into an abstract syntax tree. The tree representation is built from the token sequence according to certain observations which are associated with modification in a certain order. The construction is separated in several phases where each token is considered as a node in a tree. Each phase traverses the tree many times to perform rehanging actions in order to create subtrees of certain structure. The parsing process is similar to a pipeline of methods which operates on the tree. The pipeline is quite large, and to gain transparency, a visitor pattern from JavaCC and JTree is used. The pattern is a common way to separate rehanging operations on a tree from each other [16].

4.1.2.1 Bracket management
The first phase distinguishes content of all types of brackets. The tree is traversed to find all start and end brackets to push the content in between to a deeper level. The ending bracket of a certain start bracket is found by a cumulative count. The value increases on a visited left-side bracket and decreases on a right-side bracket. The content in between is found when the value has reached zero. The range of tokens that is found is replaced by a new node containing all the intermediate tokens as its children. If a left bracket is found and the cumulative count never reaches zero before the token sequence ends, a syntax error is found which is reported back to the user.

4.1.2.2 Clean up
The second phase performs a cleanup of unnecessary tokens which are not of any use in order to construct a concrete syntax tree [14, page 201-202]. This phase traverses the tree and replaces pairs of line-breaks with a single line-break. One line break must be saved in order to determine if the statement should be printed or not. YouEngineer uses the same syntax as in MATLAB (removing the ending terminator) for printing. Beside the line-breaks this phase removes tabs, blank space, line and block comments.

4.1.2.3 Function declarations
The third phase finds and forms functions declarations. It searches for a token sequence in all subtrees which match the syntax of a function declaration. If one is found, the sequence is replaced by a new node which will prevent future phases to make use of tokens that already have been consumed.

4.1.2.4 Operators and precedence levels
The fourth phase finds and builds subtrees for all operators. This phase traverses the tree 15 times in order to bind operators with the same precedence levels as in C (binding one precedence level at each traversal). Each precedence level finds one or several operators in the sequence which is replaced by a matching subtree for the current computation. A precedence level is a group of operators which have the same “strength” of binding operands and the same associativity. Total 48 operators (where 43 of these belong to the C language) are search for existence.
As an example, multiplication and division have the same precedence. They are evaluated before addition and subtraction but after the power operator. Both addition and multiplication are associative while the power operator only has right-to-left associativity, and the subtraction and division operators have only left-to-right associativity. Addition and subtraction are members of the lowest precedence level (among the discussed levels). This level uses a left-to-right associativity because the subtraction operator demands it (addition works with both). In the same way multiplication and division form their own level, leaving the power operator on its own. The list below describes the precedence levels and the associativity on each level.

- **Highest precedence (Right-to-Left)**
  - Power to (Right-to-Left)
    - \( A ^ B ^ C \neq (A ^ B) ^ C \)
    - \( A ^ B ^ C = A ^ (B ^ C) \)

- **Middle precedence (Left-to-Right)**
  - Multiplication (Left-to-right / Right-to-Left)
    - \( A * B * C = (A * B) * C \)
    - \( A * B * C = A * (B * C) \)
  - Division (Left-to-Right)
    - \( A / B / C = (A / B) / C \)
    - \( A / B / C \neq A / (B / C) \)

- **Lowest precedence (Left-to-right)**
  - Addition (Left-to-Right/Right-to-Left)
    - \( A + B + C = (A + B) + C \)
    - \( A + B + C = A + (B + C) \)
  - Subtraction (Left-to-Right)
    - \( A - B - C = (A - B) - C \)
    - \( A - B - C \neq A - (B - C) \)
The building of the tree continues by pushing all the operators together with their operands from the highest precedence level to a deeper level in the tree. The next highest precedence level continues in the same way beginning from the root of the tree. When any operator is found, it will only have operands which are subtrees of operators with higher precedence. Figure 8 shows how a short calculation is transformed into a tree. The power operator with its operands is replaced first by a new node where the operands become children. Secondly the multiplication operator is found which leaves additions on its own.

<table>
<thead>
<tr>
<th>Code</th>
<th>Tokenization</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>1+2*3+4^5</code></td>
<td><code>\text{\texttt{Int\_Literal\_T\_0}}</code>  <code>\text{\texttt{Plus\_T\_1}}</code>  <code>\text{\texttt{Int\_Literal\_Z\_2}}</code>  <code>\text{\texttt{Multiply\_T\_3}}</code>  <code>\text{\texttt{Int\_Literal\_Z\_4}}</code>  <code>\text{\texttt{Plus\_T\_5}}</code>  <code>\text{\texttt{Int\_Literal\_W\_6}}</code>  <code>\text{\texttt{Power\_T\_7}}</code>  <code>\text{\texttt{Int\_Literal\_I\_8}}</code>  <code>\text{\texttt{Addition\_Op\_9}}</code>  <code>\text{\texttt{Int\_Literal\_T\_0}}</code>  <code>\text{\texttt{Addition\_Op\_10}}</code>  <code>\text{\texttt{Int\_Literal\_Z\_2}}</code>  <code>\text{\texttt{Int\_Literal\_Z\_4}}</code>  <code>\text{\texttt{Power\_T\_7}}</code>  <code>\text{\texttt{Int\_Literal\_I\_8}}</code></td>
<td><code>\text{\texttt{Trace\_The\_Statement}}</code>  <code>\text{\texttt{Addition\_Op\_9}}</code>  <code>\text{\texttt{Addition\_Op\_10}}</code>  <code>\text{\texttt{Multiply\_Op\_3}}</code>  <code>\text{\texttt{Int\_Literal\_Z\_2}}</code>  <code>\text{\texttt{Int\_Literal\_Z\_4}}</code>  <code>\text{\texttt{Power\_T\_7}}</code>  <code>\text{\texttt{Int\_Literal\_I\_8}}</code></td>
</tr>
</tbody>
</table>

Figure 8. Example of a simple calculation transformed to a tree.

4.1.2.4.1 Comma as an operator
The last precedence level in the compiler is a special one. It manages comma as an operator, adding the left and right side arguments to a list. If the left argument already is a list, the right operand is appended at the end to the list of the left side argument.

*Argument declaration lists* would be an issue if function declarations were not parsed before the comma operator. *Argument declaration lists* consists of a repeating sequence of a type, id and a comma where the id of the first argument and the type of the second argument would be united in a list during this phase if the token sequence were not changed earlier.

4.1.2.5 Language check
The fifth phase is the most language specific phase. It creates subtrees for variable declarations, function declarations (reshaped), function calls, if-statements, for-statements and others. The tree is restructured by visiting different productions in the language. Each production uses a defined *look ahead* in order to determine if the sequence matches the production or not.
4.2 Type Checking

Type checking is the process of finding return types of all operations in order to select suitable instructions and detect errors regarding inconsistent type usage. Literals have predefined types but the type of variables and function calls must be derived from its declaration. In order to type check code which contains calls to functions which are declared after the point of call, the process is divided into two phases. The first phase creates scope containers to every node that indicates new scopes in the tree (e.g. entering the body of an if-, for-, or while-statement). Each declaration that is found adds its name, return type and possible argument configuration to the current scope.

The second phase traverses the tree once again and performs lookups in the current scope container to determine types of variables and function calls. The scope container itself contains a reference to the parent scope in order to access earlier declarations. The parent reference path ends with a root scope that is preloaded with namespaces, types, functions and fields. The search which performs the binding process traverses the tree in a bottom-up-order which guarantees that types of operands are decided before return-types of operations. Each node which is associated with a type saves the type in the tree. All nodes indicating a variable saves a reference to a *data area identifier* which holds information about where the variable will be initialized in memory at runtime. Function calls to user defined functions save a reference to the body of the implementation (node in the tree). Calls to predefined functions save a function pointer.
4.2.1 Operands and evaluation of constant expressions

The return type of an operator is decided by the type of the first operand. The type checker makes use of lookup tables (one for each data structure) to select a particular operator depending on the type of the second operand (if any). All operator candidates are predefined for each data structure and a certain operator (like addition) can have several overloads that match operands of different types.

All constant expressions are evaluated at compile time and are initialized and operate with the most generous data type of its kind. In the case when at least one operand is a constant and no suitable operator candidate is found because the type of the constant, an implicit cast is performed. If any overflow exception is caught during this cast, it is reported back to the user as a compilation error.

Figure 9 shows an example of a type checked tree for a part of a program. The tree demonstrates constant evaluation at @29 and @33. The operands of the multiplication which has been replaced by a constant at @29 yields an int4096 type which has been shrinked to int64 because of the variable at @39 and operator at @37.

```plaintext
int64 a = 7;
int64 b = 1 + 2 * 3 ^ 2 * a + 1;
c = (11 == b);
```

Figure 9. Example of a type checked tree.
4.2.2 Selection by Reference

Data structures in YouEngineer are able to contain static and non-static members. The selection by reference operator allows the user to access these members, either on types (static) or on instances (non-static). When the operator is used on an instance, the type of the instance is found by searching for a non-static member which has the same name as the right side argument.

A subtree of selection by reference operators is built in the same way as a chain of additions, which has its left-most operator at the deepest level in the tree. Visiting the subtree in a bottom-up order makes it possible to “bubble up” type information to bind methods and properties for each level. When the type checker comes across a selection by reference operator, it saves a list of pointers to all members available in the type of the first argument that match the name of the second argument. The list contains multiple matches if and only if the match is a function with several argument configurations. Otherwise the list contains only one pointer or none.

Figure 10 shows a tree of several selections by reference operators where a variable “a” is dereferenced. In the example, variable “a” is found to be of type “float64” in the global scope at @14 in the tree. The selection by reference node at @15 holds all members named “Round” from the non-static member space of the data structure “float64”. The “function call” operator at @15 searches the result for a candidate with one argument. The search finds a match which takes an “int64” and returns a “float64”.

```
a = 2.653245;
a.Round(4), Round(3), Round(2)
```

Figure 10. Example of a tree of several “element selection by reference” operator.
4.3 Code intelligence generation

The sequence of tokens which is generated during the tokenization phase is represented as nodes in a flat tree. The tokenization phase store a string copy from the code in each node together with its position of initialization. This information is preserved more or less during all phases which make it easy to report the origin of errors as they are discovered.

When the tokenization phase is finished, an aside saved copy of the tree is stored in order to preserve an exact source code representation. The copy is not only a copy. It creates a pointer for each node in the aside saved tree to the corresponding node in the original tree. The original tree is forwarded to the next phases which removes, reshapes and adds new nodes to the tree in order to continue with the parsing.

The aside saved tree is used by the code coloring phase which begins after the type checker. The code coloring regenerates the code with colors based on information available in the node of the tree which has been type checked. The code is colored in six different colors in order to distinguish operators from contextual keywords, types, variables, methods and comments.

The auto completion feature takes advantage of the aside saved tree as well. When the cursor moves around in the code, it is easy to find the node which matches the same position in that sequence of nodes. Each node which has a reason to reveal information to the auto completion feature has its pointer to the type checked tree intact. A strategy that works well in order to fill the auto completion list with valued information is to search the type checked tree upwards until a scope container is found. The scope container contains all members with their names and types of the current scope, which represent the main source of content. The scope container does also contain access to all other members which were declared in previous scopes through its parent pointer mentioned in 4.2.
4.4 Intermediate Representation

The type checked tree is translated to an intermediate representation (IR) which consists of an
interpretable tree for the CPU part and PTX assembler for the GPU part. The tree is generalized
by replacing specific operators (like addition and multiplication) with a common node. Calls to
predefined functions are separated from calls to user-defined functions by naming nodes
differently. Predefined functions use the pointer derived during type checking to access the
right procedure at runtime. User defined functions save a reference to the subtree within the
IR-tree that represents the body of the function. The body itself is cut and moved outside the
path of the program while the remaining nodes represent the main function in which the CPU
part launches.

4.4.1 PTX generation

Functions tagged with a “gpu” keyword are translated to PTX assembler. This is done after the
type checked tree has been generalized into an IR-tree. The assembling is done by visiting the
body of the function with visitor patterns that perform lookups to yield predefined assembler
translations for operators and functions. The function pointers in the IR-tree serve as keys to
map certain procedures to the right assembler representation. A register handler appends
register declarations to the PTX code and reuse temporary registers when possible. The
translation process makes use of the register handler in order to glue incoming and outgoing
values together when the assembler is built (moving from one level to another in the IR-tree).
The register handler stores operands and arguments either as constants or register identifiers
on a stack. A translation method for a procedure pops its arguments from the stack, appends
necessary assembler instructions, recycles used registers and pushes an identifier of the
register holding the return value on the stack.

Registers representing user named variables are occupied during the lifetime of the gpu
function. Those registers are not recycled as soon as they could be if a register liveness analysis
is performed [14, page 608-609]. The amount of registers in use when a translation procedure
returns is the initial count minus the number of registers not bound to a user named variable
among the incoming arguments, plus one if the procedure has a return value. The final text
based assembler is saved in the body of the gpu function in the IR-tree in order to be accessed
by the runtime later.

4.5 Runtime

The runtime launches the execution on the CPU side by walking the program path of the IR-
tree. The tree is explored by recursive calls while arguments, operands and return values are
pushed and popped on a stack. GPU declared kernels are invoked by launching execution of
PTX assembler through the NVIDIA driver API.

The runtime only launches the execution if no errors are found during the compilation phase.
The runtime runs in its own thread independently from the user interface and the compilation
process. The interface will terminate the current execution if the user types code faster than
the engine is possible to execute in order to work on the latest revision. When the runtime has
finished (end of program or termination) all allocated resources are freed and the GPU driver
environment is restarted if unrecoverable errors occur.
5 Results of evaluating the IDE

This chapter describes how YouEngineer performs with respect to user experience, compilation time, run time and response time.

YouEngineer is evaluated by implementing three small programs in order to experience and measure the rapid code iteration of the environment. The first two programs will run in YouEngineer only and are discussed briefly. The third program is implemented in CUDA and C++ as well in order to gain comparability regarding compile time, run time and response time.

5.1 Drawing a 3D-surface

The first program draws a 3D surface which is described by a function $z(x, y)$ that is evaluated in multiple points and saved to a vertex array on the device. The data is copied to main memory and converted to a mesh which is sent to the canvas by the CPU-side runtime.

For a small surface (about 12 000 points) the IDE is able to recompile, run and present the graph without any noticeable delay. The surface function can be changed and the user experience is that everything happens live. The compile time is about 1.2 milliseconds and the run time finishes within 40 milliseconds. It’s almost as the result of the change was pre-evaluated and waiting to be visible.

The resolution of the surface is to the largest extent limited by the conversion from GPU evaluated points to a mesh. The conversion is single threaded, backend implemented on the CPU and the canvas itself is far from optimized. Evaluating the surface and copying data from the device represents a small part of the consumed time. Figure 11 shows a screen copy of YouEngineer drawing a surface. The code used to obtain the surface is available in the Appendix in 8.2.

![Figure 11. Example of YouEngineer plotting a 3D surface of a GPU evaluated function.](image-url)
5.2 Image filtering
The second program filters out a green laser line from a laser pointer in an image which is used by a robot to navigate in a maze. The difficulty of the task is to find a suitable threshold for brightness and greenness to highlight the laser line and avoid highlighting reflections. The program is presented as an example of a situation where a standalone task needs to be calibrated to work at all. The program uses a GPU evaluated kernel that works as a filter to cancel out unwanted areas of the image.

YouEngineer allows the user to observe the result from a modified threshold (or any other part of the code) with a direct response. The compilation time is below 4 milliseconds and the run time is about 70 milliseconds for an image of size 640x480. Figure 12 shows a screenshot of YouEngineer highlighting the laser line.

Figure 12. Example of YouEngineer highlighting the laser line in a image from a GPU evaluated filter.
5.3 RSA encryption/decryption

The third program to be demonstrated performs RSA encryption/decryption of an image. This example is heavier with regard to computations than memory usage in comparison to the previous examples. The heavier computation will demonstrate a scenario where the GPU has potential to allow rapid code iteration, when the CPU cannot. This example, running in YouEngineer is the essence of this thesis. It enhances the idea of importance to have an IDE with rapid code iteration that includes the GPU in the picture. This subject will be resumed in the next chapter about benchmarking YouEngineer towards Visual Studio using C++ with SDL-threads, and Visual Studio using CUDA.

The RSA encryption/decryption program loads an image from disk, converts it to an RGB-byte array, invokes encryption on the GPU, prints the encrypted data as an image, invokes decryption (on the GPU again) using the encrypted data, and prints the decrypted result as an image. The kernel operates with a one dimensional mesh (configuration of blocks and threads), where the size of the mesh depends on the number of bytes to encrypt/decrypt. The kernel is looking at a small window of the data at the time. The kernel loads a few byte from the input array (which is shared among all CUDA cores) and represents it as a number which serves as the RSA message. The transformed message is then saved to an output array without interference with other kernels, which make the computation embarrassingly parallelizable. The encrypted/decrypted is computed using a user implemented modular exponentiation operator.

Figure 13 shows a screen copy of YouEngineer running RSA encryption/decryption. The code implemented in YouEngineer is available in Appendix at 8.3.

Performing changes in the code of the RSA example in YouEngineer yields an immediate response for a small image of 300x225 pixels (192 KiB of data). The compilation time is about 13 milliseconds for the entire program which is about 200 lines of code. The total runtime takes less than 70 milliseconds, which includes the time it takes to load the image from disk, copy it back and forth from the graphics card, and make it visible in the canvas.

If the resolution of the image is increased to 2560x1600 (11.625 MiB of data) YouEngineer hits the limit of what a generous audience can call an immediate response. The whole cycle from a change in the source code to a visible result is in this case about 310 milliseconds.
Figure 13. Example of the YouEngineer IDE running RSA encryption/decryption.
5.3.1 RSA implementation details

RSA [17] is a simple algorithm which is nothing but a modular exponentiation operator, [18] that can be implemented easy (at least with defeat for speed). The used implementation restricts arithmetic operations to 64 bits. The RSA-key itself can only be half that size without causing arithmetic overflow (multiplication of 32 bits yields a 64 bit result). In real world applications the size of the key is much larger (typically 512-4096) in order to guarantee protection. The reason to use such small key size in this example is to avoid implementing support for big numeric arithmetic (which must be identical in multiple environments to perform the benchmark).

Figure 14 shows the modular exponentiation operator used in this example. The algorithm takes logarithmic time with respect to the size of the exponent to finish the calculation. The last division operator (’/ at line 10) is an integer division operator (YouEngineer returns a float value when the traditional division operator is used).

```c
uint64 powermod(uint64 base, uint64 exponent, uint64 modulus) {
  uint64 result = 1;
  while (exponent > 0) {
    if ((exponent % 2) == 1)
      result = (result * base) % modulus;
    base = (base * base) % modulus;
    exponent = exponent / 2;
  }
  return result;
}
```

Figure 14. Example of the powermod operator that is used to encrypt and decrypt.

Figure 15 shows how the modular exponentiation is invoked in order to obtain encryption and decryption. The unsigned integer numbers N and E are public RSA keys which are used to encrypt the message while D is private and used with N for decryption.

```c
uint64 N = 1076560937;
uint64 E = 53;
uint64 D = 914005457;
uint64 msg = 7;
msg_encrypted = powermod(msg, E, N);
msg_decrypted = powermod(msg_encrypted, D, N);
```

Figure 15. Example of how the powermod operator is used to encrypt and decrypt data with RSA.
5.4 Benchmark
To obtain an idea of how YouEngineer scales compared to CUDA or a fully utilized CPU equivalent using C++ with SDL-threads, the RSA encryption/decryption program from chapter 5.3 is implementation for comparison purpose in these environments as well. Below is a list of candidates which are compared in the following benchmark:

- **C++ 1T** - A traditional C++ implementation targeting the CPU using a single thread. The candidate is compiled from an independent, minimalistic C++ project using Visual Studio 2010.

- **C++ SDL 32T** - A traditional C++ implementation targeting the CPU using 32 threads (best achieved performance). The candidate is compiled from an independent, minimalistic C++ project using Visual Studio 2010 with SDL version 1.2.14.0.

- **CUDA** - A traditional CUDA (version 5.0) implementation targeting the GPU with bytecode in ordinary manner compiled from an independent, minimalistic CUDA template project using Visual Studio 2010.

- **YouEngineer** - The YouEngineer implementation described in 5.3 targeting the GPU JIT-compiler through a DLL that reveals the CUDA API.

- **C#/PTX** – A C# .NET implementation using pregenerated PTX assembler obtained from verbose output when compiling the CUDA candidate, with the NVIDIA CUDA compiler. The PTX assembler is passed as a parameter from C# to a DLL that reveals the CUDA API in order to launch the execution (same DLL used by YouEngineer).

Figure 16 shows a bar graph of the timings divided on compile time and run time for the different candidates where 11.625 MiB of input is encrypted and decrypted.

![Response time](image)

**Response time**

<table>
<thead>
<tr>
<th></th>
<th>Compile time</th>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>YouEngineer</td>
<td>13</td>
<td>138</td>
</tr>
<tr>
<td>C#/PTX</td>
<td>270</td>
<td>196</td>
</tr>
<tr>
<td>CUDA</td>
<td>2100</td>
<td>276</td>
</tr>
<tr>
<td>C++ SDL 32T</td>
<td>950</td>
<td>690</td>
</tr>
<tr>
<td>C++ 1T</td>
<td>950</td>
<td>3816</td>
</tr>
</tbody>
</table>

Figure 16. Response time among the different candidates.
The implementation in YouEngineer is about 200 lines of C-like code, and the other candidates project specific code is about the same size. But the other candidates contain includes which make their amount of code compiled much larger.

The exact difference in response time between the candidates is hard to measure because YouEngineer launches the compile and run command immediately when a change in the code occurs. The other candidates are launched from Visual Studio in ordinary manner, which requires a manual run-and-build command to be triggered. To facilitate the comparison, the difference in how the “compile and run” commands are triggered is ignored. But, the impact on user experience by avoiding the need to manually hit the compile button should not be neglected.

The response time in figure 16 is measured as the compile time, plus the time it takes to run the program, where the time to render the image and load it from disk is excluded (equal in all candidates). The smaller compile time in YouEngineer is reached by compiling less code of a small language and using just-in-time compilation instead of generating native assembler.

The smaller run time in YouEngineer compared to CUDA is reached by avoiding a high initiating cost of establishing a new CUDA runtime environment at each code-iteration. Programs running in YouEngineer are recycling the same CUDA runtime environment between code changes instead of demanding a new environment like a launch of an ordinary new compiled program dose. YouEngineer have the CUDA initiation cost too, but it only occur when the IDE is started.

Another observation which is a bit unexpected is the shorter run time of the C#/PTX candidate compared to CUDA. The C#/PTX candidate is running the same program as the CUDA candidate, but as verbose PTX passed for JIT compilation instead of running a CUDA binary. Why the shorter run time is observed is unknown. A discussion about the phenomenon is addressed in the discussion section after covering more details about the run time of this benchmark.

5.4.1 Compile time details
The 200 lines of code (from the RSA implementation, section 5.3) in YouEngineer correspond to 7700 characters. These characters are transformed to 2000 tokens, which in their turn are transformed to an abstract syntax tree. A small part of that tree represents the kernel to be executed on the GPU which is translated to 150 lines of PTX assembler. Figure 17 together with table 1 show details about the time spent in the different compilation stages (which were presented during section 4) to compile the RSA example.
Figure 17. The partition of the most time consuming sections during compilation in YouEngineer.

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile time</td>
<td></td>
</tr>
<tr>
<td>Tokenization</td>
<td>4 045</td>
</tr>
<tr>
<td>Language (building abstract syntax tree)</td>
<td>3 123</td>
</tr>
<tr>
<td>Dismembering brackets</td>
<td>243</td>
</tr>
<tr>
<td>Cleaning token sequence</td>
<td>356</td>
</tr>
<tr>
<td>Finding declarations</td>
<td>123</td>
</tr>
<tr>
<td>Operator tree building</td>
<td>2 130</td>
</tr>
<tr>
<td>Visiting productions</td>
<td>137</td>
</tr>
<tr>
<td>Rest</td>
<td>132</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>2 162</td>
</tr>
<tr>
<td>Scope analysis</td>
<td>128</td>
</tr>
<tr>
<td>Constants initiation</td>
<td>439</td>
</tr>
<tr>
<td>Constant arithmetic</td>
<td>251</td>
</tr>
<tr>
<td>Type checking</td>
<td>1 344</td>
</tr>
<tr>
<td>Rest</td>
<td>1</td>
</tr>
<tr>
<td>IR generation</td>
<td>2 380</td>
</tr>
<tr>
<td>PTX generation</td>
<td>2 065</td>
</tr>
<tr>
<td>Rest</td>
<td>316</td>
</tr>
<tr>
<td>Code intelligence mapping</td>
<td>402</td>
</tr>
<tr>
<td>Code coloring</td>
<td>14 877</td>
</tr>
<tr>
<td>Rest</td>
<td>808</td>
</tr>
<tr>
<td>Post cleanup</td>
<td>15 121</td>
</tr>
</tbody>
</table>

Table 1. Detailed timings of the compilation process in YouEngineer.
Figure 17 show that the largest section of time during the compilation stage is spent with tokenization. The next largest part is the language part of constructing an abstract syntax tree, where managing the precedence levels take most of the time (“Operator tree building” in table 1). The third most time consuming section is the preprocessing where type checking is the most time consuming part. When these three sections are done, the “code intelligence mapping” is started in order to color the code and generate content for the autocomplete feature.

Regarding the Visual Studio candidates in the benchmark, CUDA and the C++ SDL projects compile time depends much on the magnitude of the change. The compile time can be very large (18 seconds was observed for recompilation of the CUDA candidate) even for a minimalistic project. However a minor change like changing the value of a variable in one file can be carried out quite quickly (2.1 second as presented for CUDA in figure 16). The compile time which are visible in figure 16 for the CUDA and C++ candidates are the smallest that could be observed after changing a value of one variable. The compile time presented for YouEngineer is independent from the magnitude of the change. YouEngineer performs a full recompilation of the code for each change, and does not perform any caching of intermediate representations that corresponds to unchanged blocks of code.

An issue regarding compile time measurements is that Visual Studio will not display compilation time of C# projects. Visual Studio keeps track of changes that are made to avoid recompilation of entire projects through a background worker that process C# code after each change. This behavior in Visual Studio is good because it reduces the response time, but the lack of possibility to measure it complicates the benchmark. The only substitute is to use the time from a complete recompilation using a windows batch command to MS build. Which is sad because the MS build time is likely larger than the build time within Visual Studio. The reader should be aware that the compile time presented for the C#/PTX candidate in figure 16 is the full MS build time.

5.4.2 Run time details
The run time section in figure 16 reveals a difference in speedup between encryption and decryption if the number is intersected into more detailed components. Encryption in YouEngineer is about 3.3 times faster than utilizing the CPU in C++ with SDL threads. The corresponding speedup for decryption is about 5.8 times. Both encryption and decryption address the same amount of memory, but decryption spends more time within the modular exponentiation operator. Computational intensity is a decisive threshold for best choosing between the CPU and the GPU. A program with too low computational intensity will not benefit from the GPU. Such programs make it possible to achieve better response time in any IDE with rapid code iteration utilizing the CPU only. Figure 18 and 19 show bar graphs of timings for the run time part of the response time for encryption and decryption as if they were two independent programs. If the timings are compared with the numbers in figure 16, the "Initiating CUDA" cost should only by counted once. But, the cost is presented in both figures 18 and 19 in order to visualize its size in relation to different sizes of computation times.
Figure 18. The most distinct time consuming parts during execution of encryption.

Figure 19. The most distinct time consuming parts during execution of decryption.
The reason for a longer decryption computation relies in the power modular operator algorithm, together with the values of the RSA keys E and D (see figure 14 and 15). The while-loop (figure 14) takes $\log(E) = 6$ iterations to complete when decryption takes $\log(D) = 30$ iterations.

The bar graphs in figure 18 and 19 consist of several timings which are stacked on top of each other. The lowest field visualizes the time it takes to initialize a CUDA environment before any computation or memory copy can take place. The second field (from the bottom) visualizes the memory allocation time for two arrays (input and output). The third field visualizes the copy time of input data to the device. The fourth field visualizes the execution time of the kernel, and the last (fifth) field visualizes the copy time of output data back to main memory from the device.

The “Initiating CUDA” cost only occurs once if encryption and decryption is running successively. But the cost is represented in both figure 18 and 19 in order to visualize its impact on the run time depending on the computational intensity of the program.
6 Discussion

This chapter brings up reflections about YouEngineer, rapid code iteration and the benchmark.

An IDE designed for rapid code iteration benefits from recycling resources between incremental compilations/evaluations instead of considering every code iteration independent from the previous. YouEngineer achieves 40% faster run time by recycling the CUDA runtime environment instead of creating a new one for each change in the code (compare run time of YouEngineer and the C#/PTX candidate in figure 16). If the initiation cost continues to be as high as 80 milliseconds in the future, it will prevent IDEs to benefit from the GPU for rapid code iteration if the CUDA environment must be reinitiated for each code change. It will at least be the case for less computational intense programs. The statement is emphasized by looking at an almost draw between the CPU candidate and the GPU candidates having the initiation cost in figure 18.

The CUDA environment is recycled at an application level in the runtime of YouEngineer. Visual Studio launches applications by calling the operating system that creates a new process for each launch which forces resources to be reallocated (to ensure process integrity). The design of testing a new program in a new process from a call to the operating system is not unique to Visual Studio. It is the only way to start a program running natively without a runtime, and the C++/CUDA candidate in the benchmark is an example of a native program.

Other project templates associated with a runtime in Visual Studio are not enforced with a launch restriction of using the operating system as a middle man. Visual Studio can theoretically choose to launch C# applications by interacting directly with the .NET runtime environment instead of calling the operating system. The gain of such interaction can be an agreement on decreased application integrity between two launches of the same program. Such action can reduce overhead costs and reduce delays. If Visual Studio, the .NET runtime and the C# language evolves in such direction, with GPU support, it can be a comprehensive platform for rapid code iteration.

6.1 Fast response

YouEngineer reaches a faster response time than CUDA with Visual Studio. The main reason is the compile time which is decreased to 0.62%. The speed increase of 160 times does not mean that YouEngineer compiles faster than NVCC or any other compiler. It means that YouEngineer compiles much less code, of a smaller language with less language features, to a representation further away from native. Compiling C into native assembler is more time-consuming work than generating an interpretable tree structure like YouEngineer does.

The compilation in YouEngineer does not perform beneficial operations to improve incremental compilation on source code which is typed by the user. The faster response is achieved from the creation of the surrounding context (included types and procedures) which is inherited instead of rebuilt (from header files) at each change. The managing of header files is a big issue when it comes to achieving fast compilation with C/C++. Both Microsoft and the GNU compiler address the issue by supporting a representation for precompiled headers in order to avoid the need to reparse unchanged files for each compilation unit. This can reduce
the load on the preprocessor significantly. But attempts to get precompiled headers to work with CUDA have been unsuccessful, and it is known to be a struggle if it is possible at all. Searched documentation on precompiled headers with Visual Studio and CUDA has not mentioned the topic at all.

6.1.1 PTX versus NVCC compiled CUDA binaries
The 50% decrease in execution time in YouEngineer compared to CUDA during decryption was a surprise (compare the values of “Execute kernel” in figure 19). The PTX assembler which is generated by YouEngineer and NVCC (switching on verbose output) only reveals two small differences in the RSA implementation. The first one is that CUDA calls the powermod operator as an independent function. This gives rise to an extra branch since NVCC decides to keep the function call and not perform an inline expansion. The second difference is that CUDA calls the “memcpy” function from the C library to load and store the RSA message (four bytes of data) to global memory. YouEngineer uses a load and store procedure that performs an inline expansion which is shorter and does not give rise to any branch. The difference in implementation makes the PTX assembler generated by NVCC (available in Appendix at 8.6) a bit larger than the PTX produced by YouEngineer (230 versus 160 lines) (available in Appendix at 8.4 and 8.5). But the differences are cheap pre and post tasks of the much heavier modular exponentiation calculation. They should not make impact on the execution speed.

The phenomenon was a mystery until the NVCC generated assembler from the CUDA project was passed for JIT compilation in the same manner as in YouEngineer. Executing the verbose PTX output from CUDA yields the same speed as in YouEngineer which is twice as fast as the CUDA binaries. The "C#/PTX" candidate in the benchmark uses the verbose output while the "CUDA" candidate uses the binaries. The experiment shows that both PTX versions are comparable in speed. But why the JIT compiled PTX runs faster than the CUDA binaries is still a mystery. One guess is that the NVIDIA driver performs more or better optimizations during JIT compilation compared to what NVCC does at the compilation stage. But, that is only a guess.

6.2 About the compiler design
The method used to parse code in YouEngineer differs from a well-established design pattern that is used by general compile tools like JavaCC (Java compiler compiler) or JACC (Just another compiler compiler). These tools can build a compiler that compiles a provided language to an abstract syntax tree from a sequence of characters.

In an earlier project I learned how JavaCC together with JITree can be used to compile the MiniJava language [19] to Jasmin assembler [20]. JavaCC builds the tree by iterating the token sequence once. It performs a search in order to match the entire token sequence to a provided syntax which has different restrictions on lookahead [21] for each production [14, page 42-44]. The practical approach is that several syntax productions are visited like visiting branches in a tree to search a match for the entire code. When the match is found the parse tree is obtained from the selected path of visited (or opened) productions in the language.

YouEngineer iterates the token sequence several times instead of once where each time is less of a search and more of a certain modification on the sequence. It is hard to say which
approach is best. But YouEngineer allows parsing code in parallel from an early stage. This is true for JavaCC as well by parsing large blocks of the language like classes independently from each other, or visiting several productions simultaneously. But for a smaller language with less productions, and no classes (like the language used by YouEngineer) the parallelization is harder to apply with the strategy used in JavaCC.

YouEngineer can branch the compilation when all brackets has been grouped which occur in the first phase of tree building. Individual threads can parse the content of each bracket independently of each other until it is time for type checking. When the type checker has covered the first scope it is possible for parallel threads to continue with all scopes on the next level. However, these are optimizations that not are tested. Today YouEngineer performs parsing and type checking in a single thread. Parallelization and optimizations in general have potential to improve the performance of YouEngineer further.

As an example, the tokenization phase can split the source code on a whitespace character (outside of a string literal) into a desired number of buckets to be tokenized independent from each other, in parallel. Doing that has potential to improve the overall performance since the tokenization phase takes 30% of the compile time itself (see figure 17). Experimenting with such improvements is hard to do if a general compile tool like JavaCC generates the compiler in a standardized way. When the design decision for YouEngineer was made, it was unknown to which extent parallelization was needed in order to achieve a compile time small enough to provide a direct response. Saving the possibility to perform such optimizations on YouEngineer was one reason why no general compile tools were used.

6.3 Future development
There are a large number of features that can improve the usability and scalability of YouEngineer or a similar environment. The most limiting elements are the lack of support for data types and PTX translations. The YouEngineer implementation supports what is used in the benchmark, but not much more. The backend must be extended to be useful in a wider range of scenarios.

6.3.1 Infinite kernel loop instability
A highly prioritized feature (or safety measure) is to avoid NVIDIA driver instability when infinite loops are typed in GPU kernels. The driver (and the operating system) has a maximum default value on GPU run time of two seconds for a single process. When the two seconds has passed the driver is reloaded which terminates all interaction with the physical graphics card. The screen will freeze and all applications (not only YouEngineer) with an assigned GPU environment must abort and reallocate its resources. This is very disturbing since unfinished code can eventually match the language without errors and be sent for execution, even when it was not meant to. An easy fix is to add conditional branches which depend on the clock and which exit the kernel at the beginning of each loop during assembler generation. This would prohibit executions to run infinitely long (or two seconds) when an infinite loop is expressed by accident, and at the same time allow automatic execution during typing.
6.3.2 Implementation of new language features

Rapid code iteration can benefit from an initiation phase where code is evaluated only once, or each time it is changed independently from the rest of the code. The initiation phase can operate in a scope of higher level which loads data from elsewhere and initializes constant declarations that become accessible to the rest of the program. In this way more time can be spent on executing the part of the program that actually changes.

The use of an initiation phase is a bit related to recycling of results from function calls to functions without side effects. The idea is to store return values in a cache from functions which have an intense amount of identical calls. The functions can either be picked out by the programmer or the compiler (or both) to reduce the runtime. Such support can simplify implementation of calculations that depend on dynamic programming to be evaluated efficiently.

Considering functions in general, CUDA supports individual functions to be compiled for both CPU and GPU environments. The feature makes it possible to define one function (like the modular exponentiation operator) that can be called both by the CPU and the GPU. This feature is not supported in YouEngineer, and it is quite tricky to implement. YouEngineer would have to introduce a context switch that forces such a function to inherit a section of functionality between available features in the CPU and the GPU backend. Such functions can only use types and members that exist in both worlds. At this point YouEngineer only supports calls to functions in the backend from the GPU context.

I think there are many possibilities to improve both YouEngineer and the user experience in general when targeting GPUs. YouEngineer is a small example. But it points out that the GPU is a powerful tool for rapid code iteration, even when it is working with very rapidly produced kernels. There is reason to consider the GPU as a computation target not only for improved execution speed, but as a possibility to improve user experience with rapid code iteration as well. And this was one of the main questions this thesis aimed to explore.
7 Bibliography


8 Appendix

8.1 Grammar of YouEngineer
The context-free grammar compiled by YouEngineer in Backus-Naur-form.

<code-area>
 ::= <sentence-list>

<sentence-list>
 ::= <sentence>
 ::= <sentence> <sentence-list>
 ::= ε

<sentence>
 ::= <function-definition>
 ::= <result-viewd-variable-declaration>
 ::= <silent-variable-declaration>
 ::= <work-performers>

<work-performers>
 ::= <compound-block>
 ::= <result-viewed-expression>
 ::= <silent-expression>
 ::= <iteration-statements>
 ::= <selection-statements>
 ::= <return>

<return>
 ::= "return" ';' <new-line>
 ::= "return" <silent-expression> ';' <new-line>

<function-definition>
 ::= <cpu-function-heading> <function-body>
 ::= <gpu-function-heading> <function-body>

<cpu-function-heading>
 ::= <type> <identifier> '(' <parameter-phrase> ')'

<gpu-function-heading>
 ::= 'gpu' 'void' <identifier> '(' <parameter-phrase> ')'

<parameter-phrase>
 ::= <parameter-list>
 ::= 'void'
 ::= ε

<parameter-list>
 ::= <parameter>
 ::= <parameter> ',' <parameter-list>

<parameter>
 ::= <type> <identifier>

<function-body>
 ::= <compound-block>

<compound-block>
 ::= <new-line> '{' <new-line> <code-area> <new-line> '}'<new-line>

<result-viewd-variable-declaration>
 ::= <variable-declaration> <new-line>

<silent-variable-declaration>
 ::= <variable-declaration> ';' <new-line>
<variable-declaration>
 ::= <identifier> '=' <expression>
 ::= <type> <identifier>
 ::= <type> <identifier> '=' <expression>
</variable-declaration>

<result-viewed-expression>
 ::= <expression> <new-line>
</result-viewed-expression>

silent-expression
 ::= <expression> ';' <new-line>
</silent-expression>

<expression>
 ::= <argument-list>
<argument-list>
 ::= <argument-list> ',' <direct-assignment>
 ::= <direct-assignment>
<direct-assignment>
 ::= <identifier> '=' <direct-assignment>
 ::= <assignment-by-sum>
<assignment-by-sum>
 ::= <assignment-by-sum> '+=' <assignment-by-difference>
 ::= <assignment-by-difference>
<assignment-by-difference>
 ::= <assignment-by-difference> '-=' <assignment-by-product>
 ::= <assignment-by-product>
<assignment-by-product>
 ::= <assignment-by-product> '*=' <assignment-by-quotient>
 ::= <assignment-by-quotient>
<assignment-by-quotient>
 ::= <assignment-by-quotient> '/=' <assignment-by-reminder>
 ::= <assignment-by-reminder>
<assignment-by-reminder>
 ::= <assignment-by-reminder> '%=' <assignment-by-bitwise-left-shift>
 ::= <assignment-by-bitwise-left-shift>
<assignment-by-bitwise-left-shift>
 ::= <assignment-by-bitwise-left-shift> '<<=' <assignment-by-bitwise-right-shift>
 ::= <assignment-by-bitwise-right-shift>
<assignment-by-bitwise-right-shift>
 ::= <assignment-by-bitwise-right-shift> '>>=' <assignment-by-bitwise-and>
 ::= <assignment-by-bitwise-and>
<assignment-by-bitwise-and>
 ::= <assignment-by-bitwise-and> '&=' <assignment-by-bitwise-xor>
 ::= <assignment-by-bitwise-xor>
<assignment-by-bitwise-xor>
 ::= <assignment-by-bitwise-xor> '¤=' <assignment-by-bitwise-or>
 ::= <assignment-by-bitwise-or>
<assignment-by-bitwise-or>
 ::= <assignment-by-bitwise-or> '|=' <logical-or>
 ::= <logical-or>
<logical-or>
 ::= <logical-or> '||' <logical-and>
 ::= <logical-and>
<logical-and>
 ::= <logical-and> '&&' <bitwise-or>
 ::= <bitwise-or>
<bitwise-or>
 ::= <bitwise-or> ']' <bitwise-xor>
 ::= <bitwise-xor>
<bitwise-xor>
  ::= <bitwise-xor> '!' <bitwise-and>
  ::= <bitwise-and>
<bitwise-and>
  ::= <bitwise-and> '&<logical-equal>'
  ::= <logical-equal>
<logical-equal>
  ::= <logical-equal> '==' <logical-not-equal>
  ::= <logical-not-equal>
<logical-not-equal>
  ::= <logical-not-equal> '!=' <less-than>
  ::= <less-than>
<less-than>
  ::= <less-than> '<<less-than-or-equal-to>
  ::= <less-than-or-equal-to>
<less-than-or-equal-to>
  ::= <less-than-or-equal-to> '<=<larger-than>
  ::= <larger-than>
<larger-than>
  ::= <larger-than> '>' <larger-than-or-equal-to>
  ::= <larger-than-or-equal-to>
<larger-than-or-equal-to>
  ::= <larger-than-or-equal-to> '=><bitwise-left-shift>
  ::= <bitwise-left-shift>
<bitwise-left-shift>
  ::= <bitwise-left-shift> '<<<bitwise-right-shift>
  ::= <bitwise-right-shift>
<bitwise-right-shift>
  ::= <bitwise-right-shift> '>><addition>
  ::= <addition>
<addition>
  ::= <addition> '+' <subtraction>
  ::= <subtraction>
<subtraction>
  ::= <subtraction> '-' <multiplication>
  ::= <multiplication>
<multiplication>
  ::= <multiplication> '*' <element-wise-multiplication>
  ::= <element-wise-multiplication>
<element-wise-multiplication>
  ::= <element-wise-multiplication> './<int-division>
  ::= <int-division>
<int-division>
  ::= <int-division> '/' <division>
  ::= <division>
<division>
  ::= <division> '/' <element-wise-division>
  ::= <element-wise-division>
<element-wise-division>
  ::= <element-wise-division> '/' <modulo>
  ::= <modulo>
<modulo>
  ::= <modulo> '%' <power-to>
  ::= <power-to>
<power-to>
   ::= <power-to> '^' <prefix-increment>
   ::= <prefix-increment>
<prefix-increment>
   ::= <prefix-increment> '++' <prefix-decrement>
   ::= <prefix-decrement>
<prefix-decrement>
   ::= <prefix-decrement> '--' <unary-plus>
   ::= <unary-plus>
<unary-plus>
   ::= <unary-plus> '+' <unary-minus>
   ::= <unary-minus>
<unary-minus>
   ::= <unary-minus> '!' <logical-not>
   ::= <logical-not>
<logical-not>
   ::= <logical-not> '!' <bitwise-not>
   ::= <bitwise-not>
<bitwise-not>
   ::= <bitwise-not> '~' <type-cast>
   ::= <type-cast>
<type-cast>
   ::= <type-cast> '(' <identifier> ')' <new>
   ::= <new>
<new>
   ::= <new> 'new' <delete>
   ::= <delete>
<delete>
   ::= <delete> 'delete' <suffix-increment>
   ::= <suffix-increment>
<suffix-increment>
   ::= <suffix-increment> '++' <suffix-decrement>
   ::= <suffix-decrement>
<suffix-decrement>
   ::= <suffix-decrement> '--' <function-call>
   ::= <function-call>
<function-call>
   ::= <function-call> '(' <argument-list> ')' <array-subscripting>
   ::= <array-subscripting>
/array-subscripting>
   ::= <array-subscripting> '[' <argument-list> ']' <element-selection-by-reference>
   ::= <element-selection-by-reference>
/element-selection-by-reference>
   ::= <element-selection-by-reference> '.' <parentheses>
   ::= <parentheses>
/parentheses>
   ::= <parentheses> '(' <direct-assignment> ')' <direct-value>
   ::= <direct-value>
/direct-value>
   ::= <identifier>
   ::= <constant>
<iteration-statements>
    ::= <for-statement>
    ::= <while-statement>
<for-statement>
    ::= 'for' '(' <initialization-expression> ';' <control-expression> ';' <iteration-expression> ')' <work-performers>
<initialization-expression>
    ::= <variable-declaration>
    ::= <expression>
    ::= ε
<control-expression>
    ::= <expression>
    ::= ε
<iteration-expression>
    ::= <expression>
    ::= ε
<while-statement>
    ::= 'while' '(' <control-expression> ')' <work-performers>
<selection-statements>
    ::= <if-statement>
    ::= <if-else-statement>
<if-statement>
    ::= 'if' '(' <control-expression> ')' <work-performers>
<if-else-statement>
    ::= 'if' '(' <control-expression> ')') 'else' <work-performers>
<type>
    ::= <identifier>
<constant>
    ::= token marked as either number or string
<identifier>
    ::= token marked as identifier
<new-line>
    ::= token marked as new-line
8.2 Drawing 3D surface (YouEngineer code)

gpu void foo(bytePtr globalAddress)
{
    uint64 gx = blockSize_x * blockId_x + thredId_x;
    uint64 gz = blockSize_y * blockId_y + thredId_y;

    // Size of a vertexPositionColor is 16 bytes
    bytePtr p = globalAddress + (gz * blockSize_x * gridSize_x + gx) * 16;

    float32 xf = ((float32)(gx) - (float32)(blockSize_x * gridSize_x) * 0.5) * 0.44;
    float32 zf = ((float32)(gz) - (float32)(blockSize_y * gridSize_y) * 0.5) * 0.8;

    float32 y = sin(zf * 0.4 + xf * xf * zf * -0.0088) + xf * xf * 0.008;

    globalStore_float32(p, xf);
    globalStore_float32(p + 4, y);
    globalStore_float32(p + 8, zf);
}

uint64 width = 6;
uint64 height = 8;

d = Gpu.GetMaxGFlopsDevice();
bytePtr p = dMalloc(16*width*16*height*16);

kernel = foo(p);
kernel.Run(kernel, 16, 16, 1, (int64)width, (int64)height, 1)

v = d.GetAs_VertexPositionColorList(p, 16 * width * 16 * height * 16);
plane = Shapes3D.PlaneXZ(v, (uint32) 16 * width, (uint32) 16 * height);

stage
s.Add(plane);
s.SetCameraResolution(400, 400);
s.SetCameraPosition(-59, 20.0, -59);
s.SetCameraLookAt(0, -10.0, 0);
s.UseFillModeWireframe();
s.SetRotationAngle(((float32)(pi * 0.25)));
s.SetScale(1, 1, 1);
s.SetLocation(0, 0, 0);
8.3 RSA encryption/decryption (YouEngineer code)

uint64 N = 1076560937;
uint64 E = 53;
uint64 D = 914005457;
pTimer tmpTimer;

//rgbBitmap bmpOrg = new rgbBitmap("C:\Users\Andre\Art\Exemplbilder\ATLAS-cern_300x225.jpg");
rgbBitmap bmpOrg = new rgbBitmap("C:\Users\Andre\Art\Exemplbilder\sommerlandschaft_2560x1600.jpg");

"Time to load bitmap: "+tmpTimer.Stop() +

tmpTimer.Start();
bmpOrgData = bmpOrg.GetAs_UInt8List();
"Time to convert bitmap to uint8List: "+tmpTimer.Stop() +

device = Gpu.GetMaxGflopsDevice();

tmpTimer.Start();
device.Free(deviceMalloc(1024))
"Init CUDA (first call) time: "+tmpTimer.Stop() +

int64 numberOfThreads = 512;
uint64 data_size_to_encrypt = 3 * 128 * numberOfThreads * (bmpOrgData.Length() / 3 / 128 / numberOfThreads);
uint64 msg_enc_size = bmpOrgData.Length() + bmpOrgData.Length() / 3;
uint64 data_size_to_decrypt = 4 * 128 * numberOfThreads * (msg_enc_size / 4 / 128 / numberOfThreads);

uint64 msg_enc_int = 1;

// PowerMod with base <= msg, exponent <= E, modulus <= N
while(E > 0)
{
    if((E % 2) == 1)
    {
        msg_enc_int = (msg_enc_int * msg_int) % N;
    }
    msg_int = (msg_int * msg_int) % N;
    E = E / 2;
}
globalStore_uint32(msg_enc_out, (uint32)msg_enc_int)

gpu void encryptKernel(bytePtr msg_enc_out_base, bytePtr msg_base, uint64 E, uint64 N)
{
    uint64 i = blockSize_x * blockIdx_x + threadIdx_x;
    bytePtr msg_enc_out = msg_enc_out_base + i * 4;
    bytePtr msg = msg_base + i * 3;

    uint64 msg_int = (*((uint64)globalLoad_uint8(msg))) << 0;
    uint64 msg_enc_int = 1;

    // PowerMod with base <= msg, exponent <= E, modulus <= N
    while(E > 0)
    {
        if((E % 2) == 1)
        {
            msg_enc_int = (msg_enc_int * msg_int) % N;
        }
        msg_enc_int = (msg_enc_int * msg_enc_int) % N;
        E = E / 2;
    }
    globalStore_uint32(msg_out, (uint32)msg_enc_int)
}

gpu void decryptKernel(bytePtr msg_dec_out_base, bytePtr msg_enc_base, uint64 D, uint64 N)
{
    uint64 i = blockSize_x * blockIdx_x + threadIdx_x;
    bytePtr msg_dec_out = msg_dec_out_base + i * 3;
    bytePtr msg_enc = msg_enc_base + i * 4;

    uint64 msg_enc_int = globalLoad_uint32(msg_enc);
    uint64 msg_dec_int = 1;

    // PowerMod with base <= msg, exponent <= D, modulus <= N
    while(D > 0)
    {
        if((D % 2) == 1)
        {
            msg_dec_int = (msg_dec_int * msg_enc_int) % N;
        }
        msg_enc_int = (msg_enc_int * msg_enc_int) % N;
        D = D / 2;
    }
    globalStore_uint8(msg_dec_out, (uint8) (msg_dec_int >> 8));
    globalStore_uint8(msg_dec_out+1, (uint8) (msg_enc_int >> 8));
    globalStore_uint8(msg_dec_out+2, (uint8) (msg_dec_int >> 16));
}
```c
uint8List encrypt(uint8List msg)
{
    pTimer memAllocTimer;
    pTimer copyToDeviceTimer;
    pTimer kernelEncryptRunTimer;
    pTimer copyFromDeviceTimer;
    pTimer dummyTimer;
    dummyTimer.Start();
    dummyTimer.Stop();

    memAllocTimer.Start();
    bytePtr msgPtr = device.MallocNull(data_size_to_encrypt);
    bytePtr msgEncPtr = device.MallocNull(data_size_to_decrypt);
    memAllocTimer.Stop();

    copyToDeviceTimer.Start();
    device.SetAs_UInt8List(msgPtr, msg, data_size_to_encrypt);
    copyToDeviceTimer.Stop();

    kernelEncryptRunTimer.Start();
    device.Run(encryptKernel(msgEncPtr, msgPtr, E, N), numberOfThreads, 1, 1, numberOfBlocks, 1, 1);
    kernelEncryptRunTimer.Stop();

    copyFromDeviceTimer.Start();
    encData = device.GetAs_UInt8List(msgEncPtr, data_size_to_decrypt);
    copyFromDeviceTimer.Stop();

    totTime = memAllocTimer.DurationInSeconds() + copyToDeviceTimer.DurationInSeconds() +
               kernelEncryptRunTimer.DurationInSeconds() + copyFromDeviceTimer.DurationInSeconds();

    "Encryption on YouEngineer (GPU)"
    "----------------------------------------------------------------------";
    "MemAlloc time: " + (memAllocTimer.DurationInSeconds()*1000).Round(4) + " ms"
    "MemCopy to device time: " + (copyToDeviceTimer.DurationInSeconds()*1000).Round(4) + " ms"
    "Kernel run time: " + (kernelEncryptRunTimer.DurationInSeconds()*1000).Round(4) + " ms"
    "MemCopy from device time: " + (copyFromDeviceTimer.DurationInSeconds()*1000).Round(4) + " ms"
    "----------------------------------------------------------------------"

    "Mem size host -> device: " + (data_size_to_encrypt/1024/1024) + " MiB"
    "Mem size device -> host: " + (data_size_to_decrypt/1024/1024) + " MiB"
    "Mem copy host -> device: " +
    (data_size_to_encrypt/1024/1024/copyToDeviceTimer.DurationInSeconds()).Round(2) + " MiB/s"
    "Mem copy device -> host: " +
    (data_size_to_decrypt/1024/1024/copyFromDeviceTimer.DurationInSeconds()).Round(2) + " MiB/s"
    "----------------------------------------------------------------------"

    "Total time: " + (totTime*1000).Round(4) + " ms"
    "Total throughput: " + (data_size_to_decrypt/totTime/1024/1024).Round(4) + " MiB/s"
    "----------------------------------------------------------------------"

    return encData;
}
```

Code continues on next side...
uint8List decrypt(uint8List encMsg)
{
    pTimer memAllocTimer;
    pTimer copyDataToDeviceTimer;
    pTimer kernelDecryptRuntimer;
    pTimer dummyTimer;
    dummyTimer.Start();
    dummyTimer.Stop();
    memAllocTimer.Start();
    bytePtr msgEncPtr = device.MallocNulled(data_size_to_encrypt);
    bytePtr msgDecPtr = device.MallocNulled(data_size_to_decrypt);
    memAllocTimer.Stop();
    copyDataToDeviceTimer.Start();
    device.SetAs_UINT8List(msgEncPtr, encMsg, data_size_to_decrypt);
    device.Synchronize();
    copyDataToDeviceTimer.Stop();
    kernelDecryptRuntimer.Start();
    device.Run(decryptKernel, msgEncPtr, D, N), numberOfThreads, 1, 1, numberOfBlocks, 1, 1);
    device.Synchronize();
    kernelDecryptRuntimer.Stop();
    copyDataFromDeviceTimer.Start();
    decData = device.GetAs_UINT8List(msgDecPtr, data_size_to_encrypt);
    device.Synchronize();
    copyDataFromDeviceTimer.Stop();
    totTime = memAllocTimer.DurationInSeconds() + copyDataToDeviceTimer.DurationInSeconds() +
    kernelDecryptRuntimer.DurationInSeconds() + copyDataFromDeviceTimer.DurationInSeconds();
    "Decryption on YouEngineer (GPU)"
    "-------------------------------------------"
    "Dummy time: " + (dummyTimer.DurationInSeconds()*1000).Round(4) + " ms"
    "MemAlloc time: " + (memAllocTimer.DurationInSeconds()*1000).Round(4) + " ms"
    "MemCopy to device time: " + (copyDataToDeviceTimer.DurationInSeconds()*1000).Round(4) + " ms"
    "Kernel run time: " + (kernelDecryptRuntimer.DurationInSeconds()*1000).Round(4) + " ms"
    "MemCopy from device time: " + (copyDataFromDeviceTimer.DurationInSeconds()*1000).Round(4) + " ms"
    "-------------------------------------------"
    "Mem size host -> device: " + (data_size_to_encrypt/1024/1024) + " MiB"
    "Mem size device: " + (data_size_to_decrypt/1024/1024) + " MiB"
    "Mem copy device -> host: " + (data_size_to_decrypt/1024/1024) + " MiB/s"
    "Mem copy device -> host: " + (data_size_to_decrypt/1024/1024/copyDataFromDeviceTimer.DurationInSeconds()).Round(2) + " MiB/s"
    "Mem copy device -> host: " + (data_size_to_decrypt/1024/1024/copyDataFromDeviceTimer.DurationInSeconds()).Round(2) + " MiB/s"
    "-------------------------------------------"
    "Total time: " + (totTime*1000).Round(4) + " ms"
    "Total throughput: " + (data_size_to_decrypt/totTime/1024/1024).Round(2) + " MiB/s"
    "-------------------------------------------"
    return decData;
}

imgEncData = encrypt(bmpOrgData);

uint32 encBmpWidth = (msg_enc_size / 3) ^ 0.5;
tmTimer.Start();
new rgbBitmap(imgEncData, encBmpWidth, encBmpWidth)
"Time to convert uint8List to bitmap: " + tmTimer.Stop()

imgDecData = decrypt(imgEncData);

uint32 decBmpWidth = (msg_enc_size / 4) ^ 0.5;
tmTimer.Start();
new rgbBitmap(imgDecData, bmpOrg.width(), bmpOrg.height())
"Time to convert uint8List to bitmap: " + tmTimer.Stop()
8.4 RSA encryption kernel (YouEngineer generated PTX)

```plaintext
entry encryptKernel(.param .u64 msg_enc_out_base, .param .u64 msg_base, .param .u64 E, .param .u64 N) {  
  .reg.u64 %tmpReg0;  
  .reg.u64 %tmpReg1;  
  .reg.u64 %tmpReg2;  
  .reg.u64 %tmpReg3;  
  .reg.u32 %tmpReg4;  
  .reg.u32 %tmpReg5;  
  .reg.u32 %tmpReg6;  
  .reg.u64 %tmpReg7;  
  .reg.u64 %tmpReg8;  
  .reg.u64 %tmpReg9;  
  .reg.u8 %tmpReg10;  
  .reg.u64 %tmpReg11;  
  .reg.u64 %tmpReg12;  
  .reg.u64 %tmpReg13;  
  .reg.u64 %tmpReg14;  
  .reg.u8 %tmpReg15;  
  .reg.u64 %tmpReg16;  
  .reg.b64 %tmpReg17;  
  .reg.u64 %tmpReg18;  
  .reg.u64 %tmpReg19;  
  .reg.u64 %tmpReg20;  
  .reg.u64 %tmpReg21;  
  .reg.u64 %tmpReg22;  
  ld.param.u64 %tmpReg0, [msg_enc_out_base];  
  ld.param.u64 %tmpReg1, [msg_base];  
  ld.param.u64 %tmpReg2, [E];  
  ld.param.u64 %tmpReg3, [N];  
  mov.u32 %tmpReg4, %ntid.x;  
  mov.u32 %tmpReg5, %ctaid.x;  
  mul.lo.u32 %tmpReg4, %tid.x;  
  add.u32 %tmpReg5, %tmpReg4, %tmpReg5;  
  cvt.u64.u32 %tmpReg6, %tmpReg5;  
  mul.lo.u64 %tmpReg6, %tmpReg5, %tmpReg6;  
  add.u64 %tmpReg7, %tmpReg6, %tmpReg7;  
  mul.lo.u64 %tmpReg7, %tmpReg7, 4;  
  add.u64 %tmpReg8, %tmpReg7, %tmpReg8;  
  mul.lo.u64 %tmpReg8, %tmpReg8, %tmpReg9;  
  add.u64 %tmpReg9, %tmpReg8, %tmpReg9;  
  ld.global.u8 %tmpReg10, [%tmpReg9];  
  cvt.u64.u8 %tmpReg11, %tmpReg10;  
  shl.b64 %tmpReg11, %tmpReg11, 0;  
  mov.u64 %tmpReg12, %tmpReg11;  
  mov.u64 %tmpReg13, %tmpReg12;  
  add.u64 %tmpReg14, %tmpReg13, %tmpReg14;  
  cvt.u64.u8 %tmpReg15, %tmpReg14;  
  add.u64 %tmpReg16, %tmpReg15, %tmpReg16;  
  add.u64 %tmpReg17, %tmpReg16, %tmpReg17;  
  add.u64 %tmpReg18, %tmpReg17, %tmpReg18;  
  mov.u64 %tmpReg19, %tmpReg18;  
  @!%tmpReg19 bra L_startWhile1;  
  mul.lo.u64 %tmpReg20, %tmpReg19, %tmpReg19;  
  rem.u64 %tmpReg21, %tmpReg20, %tmpReg21;  
  mov.u64 %tmpReg22, %tmpReg21;  
  mov.u64 %tmpReg23, %tmpReg22;  
  bra L_startWhile2;  
  cvt.u32.u64 %tmpReg24, %tmpReg23;  
  st.global.u32 [%tmpReg24], %tmpReg24;  
  exit;  
}
```
8.5 RSA decryption kernel (YouEngineer generated PTX)

```
.entry decryptKernel(.param .u64 msg_dec_out_base, .param .u64 msg_enc_base, .param .u64 D, .param .u64 N) {

  .reg.u64 %tmpReg0;
  .reg.u64 %tmpReg1;
  .reg.u64 %tmpReg2;
  .reg.u64 %tmpReg3;
  .reg.u64 %tmpReg4;
  .reg.u64 %tmpReg5;
  .reg.u64 %tmpReg6;
  .reg.u64 %tmpReg7;
  .reg.u64 %tmpReg8;
  .reg.u64 %tmpReg9;
  .reg.u64 %tmpReg10;
  .reg.u64 %tmpReg11;
  .reg.u64 %tmpReg12;
  .reg.u64 %tmpReg13;
  .reg.u64 %tmpReg14;
  .reg.u64 %tmpReg15;
  .reg.u64 %tmpReg16;
  .reg.u8  %tmpReg17;
  .reg.u64 %tmpReg18;
  .reg.u64 %tmpReg19;
  .reg.u8  %tmpReg20;

ld.param.u64 %tmpReg0, [msg_dec_out_base];
ld.param.u64 %tmpReg1, [msg_enc_base];
ld.param.u64 %tmpReg2, [D];
ld.param.u64 %tmpReg3, [N];

mov.u32 %tmpReg4, %ntid.x;
mov.u32 %tmpReg5, %ctaid.x;
mul.lo.u32 %tmpReg5, %tmpReg4, %tmpReg5;
add.u32 %tmpReg6, %tmpReg5, %tmpReg4;
cvt.u64.u32 %tmpReg7, %tmpReg6;
mul.lo.u64 %tmpReg8, %tmpReg7, 3;
add.u64 %tmpReg8, %tmpReg0, %tmpReg8;
ld.global.u32 %tmpReg6, [XtmpReg9];
cvt.u64.u32 %tmpReg10, %tmpReg6;

L_startWhile1:
  setp.gt.u64 %tmpReg12, %tmpReg2, 0;
  @!%tmpReg12 bra L_endWhile2;
  setp.eq.u64 %tmpReg12, %tmpReg13, 1;
  @!%tmpReg12 bra L_endIf3;
mul.lo.u64 %tmpReg13, %tmpReg11, %tmpReg10;
rem.u64 %tmpReg11, %tmpReg13, %tmpReg3;
mov.u64 %tmpReg11, %tmpReg14;
bra L_endIf3;

L_endWhile2:
  mov.u64 %tmpReg14, %tmpReg10, %tmpReg10;
  mov.u64 %tmpReg15, %tmpReg14, %tmpReg3;
  mov.u64 %tmpReg16, %tmpReg15, %tmpReg15;
  div.u64 %tmpReg15, %tmpReg2, 2;
  mov.u64 %tmpReg2, %tmpReg15;
bra L_startWhile1;

L_endIf3:
  mov.u64 %tmpReg14, %tmpReg10, %tmpReg10;
  mov.u64 %tmpReg15, %tmpReg14, %tmpReg3;
  mov.u64 %tmpReg16, %tmpReg15, %tmpReg15;
  div.u64 %tmpReg15, %tmpReg2, 2;
  mov.u64 %tmpReg2, %tmpReg15;
bra L_startWhile1;

L_startWhile3:
  shr.u64
  mov.u64 %tmpReg18, %tmpReg11, 0;
  mov.u64 %tmpReg17, %tmpReg16;
  mov.u64 %tmpReg16, %tmpReg8, 1;
  mov.u64 %tmpReg8, %tmpReg11, 8;
  mov.u64 %tmpReg17, %tmpReg18;
  mov.u64 %tmpReg18, %tmpReg17;
  mov.u64 %tmpReg17, %tmpReg19;
  mov.u64 %tmpReg19, %tmpReg11, 16;
  mov.u64 %tmpReg20, %tmpReg19;
  mov.u64 %tmpReg20, %tmpReg18;
exit;
}
```
8.6 RSA encryption/decryption (NVCC generated PTX)

```assembly
.weak .func (.param .b64 func_retval0) memcpy
(
   .param .b64 memcpy_param_0,
   .param .b64 memcpy_param_1,
   .param .b64 memcpy_param_2
)
{
   .reg .pred %p<2>;
   .reg .s16 %r<2>;
   .reg .s32 %r<3>;
   .reg .s64 %r<10>;
   ld.param.u64 %rd3, [memcpy_param_0];
   ld.param.u64 %rd4, [memcpy_param_1];
   ld.param.u64 %rd5, [memcpy_param_2];
   mov.u64 %rd9, 0;

   BB0_1:
   add.s64 %rd7, %rd4, %rd9;
   add.s64 %rd8, %rd3, %rd9;
   st.u8 [%rd7], %rc1;
   @%p1 bra BB0_1;
   bra.uni BB0_2;

   BB0_2:
   st.param.b64 [func_retval0+0], %rd3;
   ret;
}

.visible .func (.param .b64 func_retval0) PowerMod
(
   .param .b64 base,
   .param .b64 exponent,
   .param .b64 modulus
)
{
   .reg .pred %p<5>;
   .reg .s64 %r<19>;
   ld.param.u64 %rd17, [base];
   ld.param.u64 %rd16, [exponent];
   ld.param.u64 %rd11, [modulus];
   mov.u64 %rd12, 1;
   mov.b64 %rd18, %rd12;

   BB0_1:
   setp.gt.u64 %p1, %rd16, 0;
   not.pred %p2, %p1;
   @%p2 bra BB0_5;
   bra.uni BB0_2;

   BB0_2:
   rem.u64 %rd13, %rd16, 2;
   setp.eq.s64 %p3, %rd13, 1;
   not.pred %p4, %p3;
   @%p4 bra BB0_4;
   bra.uni BB0_3;

   BB0_3:
   mul.lo.s64 %rd14, %rd18, %rd17;
   rem.u64 %rd18, %rd14, %rd11;

   BB0_4:
   mul.lo.s64 %rd15, %rd17, %rd17;
   rem.u64 %rd17, %rd15, %rd11;
   div.u64 %rd16, %rd16, 2;
   bra.uni BB0_3;

   BB0_5:
   .loc 3 20 1
   st.param.b64 [func_retval0+0], %rd18;
   ret;
}
```

Code continues on next side...
visible .entry Rsa64EncryptKernel
(.param .u64 msg_enc_out_base_ptr, .param .u64 msg_base_ptr, .param .u64 rsa_E, .param .u64 rsa_N)
{
  .local .align 8 .b8 __local_depot1[16];
  .reg .b64 %SP;
  .reg .b64 %SPL;
  .reg .s32 %r<5>;
  .reg .s64 %rd<19>;

  mov.u64 %SPL, __local_depot1;
  cvta.local.u64 %SP, %SPL;
  ld.param.u64 %rd1, [msg_enc_out_base_ptr];
  ld.param.u64 %rd2, [msg_base_ptr];
  ld.param.u64 %rd3, [rsa_E];
  ld.param.u64 %rd4, [rsa_N];

  mov.u32 %r1, %ntid.x;
  mov.u32 %r2, %ctaid.x;
  mov.u32 %r3, %tid.x;
  mad.lo.s32 %r4, %r1, %r2, %r3;
  cvt.u64.u32 %rd5, %r4;
  mov.u64 %rd6, 4;
  mad.lo.s64 %rd7, %rd5, 4, %rd1;
  mov.u64 %rd8, 3;
  add.u64 %rd9, %rd5, 3, %rd2;
  mov.u64 %rd10, 0;
  mov.b64 %rd11, %rd10;
  st.u64 [%SP+0], %rd11;

  st.param.b64 [param0+0], %rd14;
  st.param.b64 [param1+0], %rd9;
  st.param.b64 [param2+0], %rd8;
  retv;
  call.uni (retval0), memcpy, (param0, param1, param2);

  ld.param.u64 %rd15, [retval0+0];
}
ld.u64 %rd16, [%SP+8];
{
  .reg .b32 temp_param_reg;
  .param .b64 param0;
  st.param.b64 [param0+0], %rd16;
  .param .b64 param1;
  st.param.b64 [param1+0], %rd9;
  .param .b64 param2;
  st.param.b64 [param2+0], %rd8;
  .param .b64 retval0;

  call.uni (retval0), PowerMod, (param0, param1, param2);
  ld.param.b64 %rd17, [retval0+0];
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}
...code continues from previous side.

```
.visible .entry Rsa64DecryptKernel
(.param .u64 msg_dec_out_base, .param .u64 msg_enc_base, .param .u64 rsa_D, .param .u64 rsa_N)
{
  .local .align 8 .b8 __local_depot2[16];
  .reg .b64 %SP;
  .reg .b64 %SPL;
  .reg .s32 %r<5>;
  .reg .s64 %rd<19>;
  mov.u64 %SPL, __local_depot2;
  cvta.local.u64 %SP, %SPL;
  ld.param.u64 %rd1, [msg_dec_out_base];
  ld.param.u64 %rd2, [msg_enc_base];
  ld.param.u64 %rd3, [rsa_D];
  ld.param.u64 %rd4, [rsa_N];
  mov.u64 %r1, %ntid.x;
  mov.u64 %r2, %ctaid.x;
  mov.u64 %r3, %tid.x;
  mad.lo.s32 %r4, %r1, %r2, %r3;
  cvt.u64.u32 %rd5, %r4;
  mov.u64 %rd6, 3;
  mad.lo.s64 %rd7, %rd5, 3, %rd1;
  mov.u64 %rd8, 4;
  mad.lo.s64 %rd9, %rd5, 4, %rd2;
  mov.u64 %rd10, 0;
  mov.b64 %rd11, %rd10;
  st.u64 [%SP+0], %rd11;
  add.u64 %rd12, %SP, 0;
  mov.b64 %rd13, %rd10;
  st.u64 [%SP+8], %rd13;
  add.u64 %rd14, %SP, 8;
  {
    .reg .b32 temp_param_reg;
    .param .b64 param0;
    st.param.b64 [param0+0], %rd14;
    .param .b64 param1;
    st.param.b64 [param1+0], %rd9;
    .param .b64 param2;
    st.param.b64 [param2+0], %rd8;
    .param .b64 retval0;
    call.uni (retval0), memcpy, (param0, param1, param2);
    ld.param.b64 %rd15, [retval0+0];
  }
  ld.u64 %rd16, [%SPL+0];
  {
    .reg .b32 temp_param_reg;
    .param .b64 param0;
    st.param.b64 [param0+0], %rd16;
    .param .b64 param1;
    st.param.b64 [param1+0], %rd3;
    .param .b64 param2;
    st.param.b64 [param2+0], %rd4;
    .param .b64 retval0;
    call.uni (retval0), PowerMod, (param0, param1, param2);
    ld.param.b64 %rd17, [retval0+0];
  }
  st.u64 [%SPL+0], %rd17;
  {
    .reg .b32 temp_param_reg;
    .param .b64 param0;
    st.param.b64 [param0+0], %rd7;
    .param .b64 param1;
    st.param.b64 [param1+0], %rd12;
    .param .b64 param2;
    st.param.b64 [param2+0], %rd6;
    .param .b64 retval0;
    call.uni (retval0), memcpy, (param0, param1, param2);
    ld.param.b64 %rd18, [retval0+0];
  }
  ret;
```
