# An Auto-Programming Approach to Vulkan

Vladimir Frolov [1,2], Vadim Sanzharov [2], Vladimir Galaktionov [1] and Alexandr Scherbakov [2]

[1] *Keldysh Institute of Applied Mathematics, Miusskaya sq., 4, Moscow, 125047, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

**Abstract**

We propose a novel high-level approach for software development on GPU using Vulkan API. Our goal is to speed-up development and performance studies for complex algorithms on GPU, which is quite difficult and laborious for Vulkan due to large number of HW features low level details. The proposed approach uses auto programming to translate ordinary C++ to optimized Vulkan implementation with automatic shaders generation, resource binding and fine-grained barriers placement. Our model is not general-purpose programming, but is extendible and customer-focused. For a single C++ input our tool can generate multiple different implementations of algorithm in Vulkan for different cases or types of hardware. For example, we automatically detect reduction in C++ source code and then generate several variants of parallel reduction on GPU: with optimization for different warp size, with or without atomics, using or not subgroup operations. Another example is GPU ray tracing applications for which we can generate different variants: pure software implementation in compute shader, using hardware accelerated ray queries, using full RTX pipeline. The goal of our work is to increase productivity of developers who are forced to use Vulkan due to various required hardware features in their software but still do care about cross-platform ability of the developed software and want to debug their algorithm logic on the CPU. Therefore, we assume that the user will take generated code and integrate it with hand-written Vulkan code.

**Keywords**

GPGPU, Vulkan API, ray tracing, code generation

## 1. Introduction

Over the past 5 years, dramatic changes have taken place in the field of GPU programming, which put researchers and developers of deep learning algorithms, computer vision and computer graphics around the world in a difficult situation: widely used and convenient cross-platform technologies such as OpenCL, OpenMP do not provide access to the latest capabilities of modern GPU (indirect dispatch, command buffers, ray tracing, texture compression, and many other). Technologies which do provide enough hardware features are proprietary (CUDA, OptiX) or difficult/laborious (Vulkan, DX12, Metal). In practice, developers have to support several variants of the same algorithm for different GPUs to achieve the desired level of performance and compatibility. Moreover, the differences in the source code (and in performance) can be dramatic. For example, the implementation of image processing can be done via compute shaders or by using the graphics pipeline with hardware alpha blending or graphics pipeline with sub-passes on mobile HW. The essence of an algorithm does not change from how exactly it is implemented on the GPU, but developers should handle different versions and work with low-level details, which usually change for different GPUs. Our goal is to preserve pure algorithmic software description, but at the same time with the ability to use any existing or future Vulkan HW features.

*GraphiCon 2021: 31st International Conference on Computer Graphics and Vision, September 27-30, 2021, Nizhny Novgorod, Russia*
EMAIL: vova@frolov.pp.ru (V. Frolov); vadim.sanzharov@graphics.cs.msu.ru (V. Sanzharov); vlgal@gin.keldysh.ru (V. Galaktionov); alex.shcherbakov@graphics.cs.msu.ru (A. Scherbakov)
ORCID: 0000-0001-8829-9884 (V. Frolov); 0000-0001-6455-6444 (V. Sanzharov); 0000-0001-6460-7539 (V. Galaktionov); 0000-0002-5360-4479 (A. Scherbakov)

## 2. Existing Solutions

The number of existing GPU and FPGA programming technologies is significant [1] (and we believe that this indicates the high relevance of the problems outlined in the previous section). It's possible to classify these technologies into several groups.

**Libraries**. Usually aimed at specific class of developers who need implementations of specific tools or algorithms. This category includes such well-known libraries as TBB, Thrust [2], CUBLAS, IPP, NPP, MAGMA, [3], [4], HPX [5] and many others. Some of them turn into rather powerful software solutions (for example PyTorch [6], TensorFlow [7] and [8]). The main drawback of libraries is their narrow focus and limited capabilities. In addition, many libraries are deliberately deprived of the ability to be cross-platform (for example, TBB, Thrust, IPP, Intel Embree), since they are released by a certain CPU or GPU manufacturer in order to promote their own hardware and that is why they are made non-portable.

**Directive-based Programming Models.** These include technologies such as OpenMP, OpenACC [9] C ++ AMP (Microsoft), Spearmint [10], OP2 [11], in works [12-13] and DVM / DVMH [14-16].

This approach has 2 key disadvantages. The *first disadvantage* is the absence of the control over the operations of copying and distributing data between the CPU and GPU, which is done automatically. This becomes a bottleneck and results in poor performance [17]. In solving this problem, interesting results were achieved in [12], where a software pipeline was used that combines the stages of execution on the CPU and GPU. However, the pipeline cannot always help and in certain situations, memory management and copying must be strictly controlled. The *second disadvantage* of this group of technologies is the extremely weak support for GPU hardware capabilities. For example, in OpenACC it is impossible to directly use shared memory, warp voting functions, there is no support for textures (which is unacceptable for high-performance and energy-efficient computer vision and computer graphics applications). Other technologies have similar problems as well.

**Skeletal Programming**. This includes such works as SkePU [18, 19], SkelCL [20], SYCL [21] and its derivatives (Intel oneAPI). These technologies target developers actively using STL-like containers in C++. The goal is to have one implementation of the algorithm in C++, which can work efficiently both on the CPU and on the GPU, or hybrid execution (simultaneously on the CPU and GPU). The main advantage over previous group of technologies is higher efficiency for some algorithms (which actively use basic skeletons or their combinations). The disadvantage is a significant deterioration in readability, code maintainability and flexibility, since the implementation of the algorithms has to fit the skeleton which leads to unnaturally looking implementations [22]. In addition, skeletal programming inherits almost all the disadvantages of the previous group. However, it still took an important step forward, since it was the first to apply algorithmic optimizations on a GPU.

**GPGPU**. This group includes CUDA and OpenCL – technologies which are employed by a wide class of professional GPU developers and can provide access to the wide range hardware capabilities of modern GPUs. Nowadays, CUDA is the dominant technology thanks to Nvidia's technological leadership. However, in addition to the lack of cross-platform functionality, it has many drawbacks: CUDA is a very heavy technology with a huge number of functionalities and versions (11 versions at the moment), which do not always work equally well on different Nvidia GPUs. Porting a software system from one version of CUDA to another is often time consuming.

To overcome dependence on the single platform several open-source implementations of CUDA were developed using OpenCL or SIMD CPU instructions [23-25], as well as AMD HIP [26]. But this approach is inherently unsuccessful, - it has a critical problem: the lack of hardware support for certain functionality in OpenCL (or other technology used as a back-end) will lead to the fact that the algorithm either won't work at all, or will work slowly (due to software emulation of this functionality), which is unacceptable for most GPGPU programming applications. As for OpenCL itself, the main drawback of this technology is the lack of support for the hardware functionality of modern GPUs. For example, the indirect dispatch [27], which wasn't included in the recently released OpenCL 3.0 standard. This functionality is critical for complex algorithms where control flow is dependent on GPU computation [28].

**Graphics APIs, Vulkan**. This includes technologies such as OpenGL4+, DirectX10 - DirectX11, DirectX12, Metal, Vulkan. GPU programming has changed a lot over the past 5 years with new

hardware features in APIs such as Vulkan, DirectX12, and Metal. Further we will consider only Vulkan, because it is a cross-platform technology, unlike DirectX12 and Metal. Unfortunately, the complexity of developing programs using Vulkan exceeds the development using CUDA or OpenCL up to 10 times [29], which is a consequence of the manual control over many hardware capabilities.

The traditional way of reducing code complexity is to create *lightweight helper libraries*, which can be general-purpose or specific for each application. However, in this case such an approach doesn't really help much, because the user still has to work with the same Vulkan entities, setting up and connecting the relationships between them. An alternative solution is to create a *heavier version of a library* or an engine that encapsulates Vulkan entities and tries to handle things automatically (V-EZ, VUH, Kompute). But this method does not greatly distinguish the engine from the work that, for example, the OpenGL driver does. This usually leads to suboptimal implementation and bugs, as the developer largely duplicates the work of the driver (or the other API). The problem is fundamental, because if someone is going to use Vulkan, they need a very specific low-level control on certain functionality that is supposed to be implemented and carefully configured in some places, but not for everything.

**Domain Specific APIs**. This group is comprised of technologies such as Nvidia OptiX [30], Microsoft DirectML [31], AMD MIOpen. OptiX is a specialized technology for accelerating GPU ray tracing applications developed in C ++. OptiX includes two key technologies: (1) Hardware-accelerated ray tracing and (2) accelerating virtual function calls using shader tables. These 2 functionalities are also available in Vulkan and DirectX12. Since it is much easier to program in OptiX (and this is the only industrial-level technology of this kind that supports C ++), almost all industrial rendering systems have switched to OptiX: Octane, VRay, IRay, RedShift, Cycles, Thea and many others. Although the latest AMD GPUs have hardware support for ray tracing, in fact there is no alternative for users to Nvidia. Analogues (such as [32, 33]) significantly lag behind solutions based on OptiX. This is an example of how programming technology combined with hardware implementation established the monopoly of one GPU manufacturer, preventing the developers of rendering systems of the ability to create cross-platform solutions which is a significant drawback.

**Domain Specific Languages (DSL)**. This group targets developers working in specific fields, for whom it is important to achieve maximum simplification and abstraction from the details of the algorithm implementation on the GPU with a good level of performance.

For example, the languages Darkroom [34] and Halide [35-37] are designed to create image processing algorithms. Cross-platform ability in Halide is achieved by implementing a large number of low-level layers, and high efficiency due to the use of optimized filter sequences. One of the key optimizations is based on the idea of reordering operations: if we consider applying two filters sequentially to an image, then often the image can be divided into regions and the entire chain of filters can be applied to each region at once: this decreases L2 cache misses.

The disadvantage of DSL is that algorithms and knowledge written in them are difficult to transfer to other areas and difficult to integrate with the rest of software that does not use a DSL. In addition, the more areas the software solution targets, the more different DSLs will need to be used and stacked, which significantly complicates the development. For example, in modern computer graphics applications, *shaders are essentially a domain-specific language*. Currently in graphics and ray tracing pipelines an algorithm is distributed over several programs (from 2 to 5), supplemented by setting the pipeline state in C ++ code. This makes the process of writing a program extremely difficult, since there is no single description of the algorithm, but instead *there are many disparate programs and configurable links between them in different places*.

**Various high-level approaches**. This group includes different unique solutions.

[38] focuses on achieving cross-platform high performance. The goal is to have a single representation of an algorithm that can be translated into an efficient implementation on various computing systems. The means of achievement is the so-called “multidimensional homomorphisms”, a formal description of a problem at a high level that allows expressing computations with parallel patterns that can be implemented in different ways on different hardware. A significant limitation of [38] is the use of OpenCL as a low-level layer which does not provide access to many of the new capabilities of modern GPUs (including mobile ones) due to the limitations of the OpenCL. Next, [38] uses extremely limited DSL, which is a significant drawback.

TaskFlow [39] proposes a model for expressing computations in the form of static and dynamic task graphs for heterogeneous computing systems. Tasks communicate with each other by streams of data along the edges of the graph through queues using the producer-consumer scheme. TaskFlow has the ability to build heterogeneous graphs (using both CPU and GPU) and then pipelined execution similar to [13]. Such a computation model is promising since it can not only be efficiently performed on CPU and GPU, but also can be used for prototyping hardware implementations on FPGA or VLSI. The disadvantage of TaskFlow is that the algorithm must be explicitly described in terms of task graphs, which is not very convenient for development and debugging.

The PACXX compiler [40, 41] uses skeletal programming ideas, but is more convenient for use in existing code. PACXX directly implements modern C ++ constructs and translates the use of STL containers into GPU buffers. Unfortunately, PACXX does not provide access to many hardware features (for example, textures) which available even in CUDA and OpenCL. Therefore, from a practical point of view, its advantages over pragmas (for example, OpenACC) are not significant.

The main purpose of Chapel [42] is to achieve cross-platform ability, so that a single description of the algorithm can work efficiently on both the CPU and GPU. In addition, it is positioned as easier to use than CUDA. For this, authors propose new language oriented towards parallel programming. But unlike Halide, it is a general-purpose language. The key disadvantage of this approach is that the user has to port a significant description of the algorithm to this new language, which is usually met with resistance in the industry because it means lack of cross-platform ability.

Taichi [43] is a programming language and an optimizing compiler oriented on applications dealing with sparse data-structures including physical simulations, ray tracing and neural networks. Taichi allows users to write high-level code using proposed language (frontend is embedded in C++) as if they were dealing with ordinary dense multidimensional arrays. The compiler then generates intermediate representation, optimizes it and generates back C++ or CUDA code. Taichi also handles memory management and uses unified memory access feature available in CUDA. The ability to target both CPUs (by using such techniques as loop vectorization) and GPUs (although only using CUDA) is a strong point if this solution. The fact that Taichi targets operations on specific data structures allows it to produce highly optimized and efficient code, but at the same time limits its potential applications. Being a DSL Taichi also shares same drawbacks, however close integration with C++ somewhat alleviates them.

Tiramisu [44] is a polyhedral compiler (that is, considering different optimization options for the same algorithm) that specializes in high-performance code generation for different platforms. At the same time, this compiler has several limitations. First, it only supports a specialized high-level language, which makes it difficult to implement in applications in other languages and increases the time spent on porting algorithms written in other programming languages. Second, it is targeted to cluster computing and has significant limitations in terms of hardware.

The clspv compiler [45] translates OpenCL kernels into an intermediate GPU representation called SPIR-V (used by Vulkan to define shader programs) and thus can be used for Vulkan development. Unfortunately, it only supports compute shaders and is currently officially in the prototype stage (although it is already relatively stable, since it has been in development since 2017).

The Circle compiler appeared 2 years ago, but it wasn't until 2020 that it became focused on GPU programming [46]. It is currently the only C ++ compiler in the world that supports the graphics functionality of modern GPUs (graphics pipeline, ray tracing pipeline, mesh shaders). But the development is still in the early stage. Circle is a traditional compiler, which itself has all the drawbacks of the traditional approach: If a developer starts using Circle as the main tool (that is, not only for shaders, but for the entire description of the algorithm), then it becomes dependent on it and assembly for any platform (including mobile systems) *is no longer possible without* Circle.

## 2.1. Conclusion on existing solutions

General purpose technologies do not support enough hardware features which hinders the performance and energy efficiency (for example, PACXX, OpenACC, DVMH, OpenCL, CUDA, TaskFlow). Low-level industrial APIs provide such support, but development on them is extremely laborious (Vulkan, Metal, DirectX12). Domain Specific Language (DSL) technologies and languages

(Halide, Optix) are a good solution for both GPUs and even other computing systems (FPGA or ASIC). However, their key disadvantage is that algorithms and knowledge implemented with them are difficult to transfer to other areas and difficult to integrate with the rest of the software that does not use a domain-specific language. There are no technologies which can achieve 2 goals simultaneously: (1) cross-platform ability and (2) accessing specific HW features, because existing solutions don't have an intermediate layer between high-level algorithm description and its actual implementation. Best results in this direction have been achieved in [43, 44, 38, 39].

## 3. Proposed solution

The proposed programming technology is not general purpose, but it considers several different fields of application and tends to be customer-oriented. At the same time, unlike, for example, Halide [35-37], it does not use domain-specific languages (DSL), but instead extracts the necessary knowledge from ordinary C++ code. One of the main advantages of our technology is that the input source code *is not extended* by any new language construction or directives. It is assumed that the input source code *is normal hardware agnostic C++* source code which in general can be compiled by any appropriate C++ compiler (with some limitations though). This significantly increases ability to cross-platform development using suggested technology.

To achieve this, we turn the concept of programming technology upside down: instead of making a general-purpose programming technology for building various software systems, we propose an extendable technology that can be customized for a specific class of applications. Therefore, we use pattern-matching to find patterns in C++ code and transform them to efficient GPU code in Vulkan. Patterns are expressed through C++ classes, member-functions and relationships between them.

Before we proceed it is important to note the difference between our technology and most existing parallel programming technologies like CUDA, Taichi, Halide and others: they extend programming language with new constructions or propose new languages in which parallel constructions map to some efficient implementation in the hardware. Our approach is the opposite. First, we do not extend the programming language, but rather limit it. Second, our patterns don't express hardware features. Instead of that, they express algorithmic and architectural knowledge. Thus, hardware features are the responsibility of the translator, not the user. There could be a lot of patterns in total, but in this work, we have implemented limited number of them. Therefore, we consider implemented patterns by examples.

### 3.1. Patterns

Patterns are divided into *architectural* and *algorithmic*. The architectural pattern expresses architectural knowledge of some part of the software. It determines the behavior of the translator as a whole and, thus, is responsible for an application area (for example, image processing, ray tracing, neural networks, fluid simulation, etc.). Algorithmic patterns express algorithmic knowledge and define a narrower class of algorithms that can have efficient hardware implementations and can be found inside architectural patterns. For example, parallel reduction, data compaction (parallel append to the buffer), sorting, scan (prefix sum), building histogram, map-reduce, etc. Now, let's consider patterns that we have implemented in the current version of our translator:

- Architectural pattern for *image processing.* This pattern provides basic GPGPU capabilities. The input source code looks like ordinary C++ code with loops in OpenMP-style except that we don't actually use directives (pragmas). Instead of that we suppose that there is certain class with a *control* and *kernel* functions (listing 1). The kernel functions contain the code that is assumed to be ported to GPU almost "as is". The control functions are the functions, which call kernel functions, and thus they define the logic of kernel launches and resource bindings.
- Architectural pattern for *ray tracing*. The goal of this pattern is to provide access to hardware accelerated ray tracing and efficiently call virtual functions on GPU. Therefore, it can be considered as a cross-platform OptiX analogue. The significant difference between this pattern and the previous one is that in the image processing pattern loops are assumed to be placed inside kernel functions

while in the ray tracing pattern they are assumed to be placed out of control functions (and thus out of kernels too). Therefore, ray tracing pattern is convenient if complex and heavy code is used for each thread or each processed data element, but the data processing loop is straightforward. The image processing pattern is convenient when number of threads (processed data elements) changes during the algorithm and inter-thread communication on GPU is actually needed for implementation. For example, if we need to resize (down sample) image, we can process small version of the image and then upscale it back.

- Algorithmic pattern for *parallel reduction.* For this pattern, we detect access to class-data variables (members of input class, fig. 1) and generate code for parallel reduction on GPU.
- Algorithmic pattern for *parallel append* of elements to the buffer and the related pattern of subsequent *indirect dispatching*. Imagine that you have a member-function which processes input data and appends some data to a vector via "std::vector.push_back(…)". Now you are going to process selected data in the other function. This time, loop counter depends on "vector.size()" and thus actual number of threads on GPU should be different: it will be known only after first kernel finishes, therefore we have to insert indirect dispatch here.

Listing 1 shows input code example and listings 2 and 3 – *simplified* output examples.

```
1.  class Numbers {
2.  public:
3.    void CalcArraySumm(const int* a_data, uint a_dataSize) {
4.      kernel1D_ArraySumm(a_data, a_dataSize);
5.    }
6.    void kernel1D_ArraySumm(const int* a_data, size_t a_dataSize) {
7.      m_summ = 0;
8.      for(uint i=0; i<a_dataSize; i++) {
9.        int number = a_data[i];
10.       if(number > 0)
11.         m_summ += number;
12.     }
13.   }
14.   int m_summ;
15. };
```

**Listing 1:** Example of input source code for the image processing architectural pattern. Calculation of sum of all positive numbers in the array. Kernel function and its dimensions (1D, 2D or 3D) are extracted by analyzing function name (*kernel1D_ArraySumm, lines 6-12*). Control function is extracted by analyzing its code: if at least one of the kernel functions is called from this function, then it is a control function (*CalcArraySumm, lines 3-5*). Any class data members which are accessed by kernels are placed in a single "class data buffer" (*m_summ, line 14*). Access for such variables is further analyzed. If in any kernel writes single variable on different loop iterations (*line 11*), then we generate parallel reduction code at the end of the shader for this variable (listing 3).

```
1.  class Numbers_Generated : public Numbers {
2.  public:
3.    virtual void SetInOutFor_CalcArraySumm(VkBuffer a_dataBuffer) { ... }
4.    virtual void CalcArraySummCmd(VkCommandBuffer a_commandBuffer, uint a_dataSize) {
5.      m_currCmdBuffer = a_commandBuffer;
6.      vkCmdBindDescriptorSets(a_commandBuffer, ... , ArraySummLayout, &allGeneratedDS[0], ... );
7.      ArraySummCmd(a_dataSize);
8.    }
9.  protected:
10.   virtual void ArraySummCmd(size_t a_dataSize) {
11.     ...
12.     vkCmdBindPipeline   (m_currCmdBuffer, ... , ArraySummInitPipeline);
13.     vkCmdDispatch       (m_currCmdBuffer, 1, 1, 1);
14.     vkCmdPipelineBarrier(m_currCmdBuffer, ... );
15.     vkCmdBindPipeline   (m_currCmdBuffer, ... , ArraySummPipeline);
16.     vkCmdDispatch       (m_currCmdBuffer, a_dataSize/256, 1, 1);
17.     vkCmdPipelineBarrier(m_currCmdBuffer, ... );
18.   }
19. ...
20. }
```

**Listing 2:** Getting some class as input, our solution generates an interface and implementation for the GPU version of the algorithms, implemented in control functions. Due to the peculiarities of Vulkan

we have to generate 2 functions for each input control function. Thus, for *CalcArraySumm* we generate two funcs: *SetInOutFor_CalcArraySumm* and *CalcArraySummCmd*. When the first one is called, it creates descriptor set for input buffer (*a_dataBuffer* in the example) and saves it to *allGeneratedDS[0]*. We have deleted input pointer parameter *a_data*. In generated code pointer parameters of control and kernel functions are not used because this time all data is on GPU and they are accessed via descriptor sets in shaders. In this example 2 different shaders were generated: the first one is *ArraySummInitPipeline* which executes loop initialization (zero sum) and the second one is *ArraySummPipeline* which executes loop body (listing 3).

```
1.  __kernel void kernel1D_ArraySumm(__global const int* a_data, __global NumbersData* ubo, ...) {
2.    __local int m_summShared[256*1*1];
3.    ...
4.    int number = a_data[i];
5.    if(number > 0)
6.      m_summShared[localId] += number;
7.    ...
8.    barrier(CLK_LOCAL_MEM_FENCE);
9.    m_summShared[localId] += m_summShared[localId + 128];
10.   barrier(CLK_LOCAL_MEM_FENCE);
11.   m_summShared[localId] += m_summShared[localId + 64];
12.   barrier(CLK_LOCAL_MEM_FENCE);
13.   m_summShared[localId] += m_summShared[localId + 32];
14.   m_summShared[localId] += m_summShared[localId + 16];
15.   m_summShared[localId] += m_summShared[localId + 8];
16.   m_summShared[localId] += m_summShared[localId + 4];
17.   m_summShared[localId] += m_summShared[localId + 1];
18.   if(localId == 0)
19.     atomic_add(&ubo->m_summ, m_summShared[0]);
20. }
```

**Listing 3:** Example of generated shader for the clspv compiler. The original loop body transforms to lines 4—6. It can be seen that access to *m_summ* was rewritten to *m_summShared[localId]* which is further used in parallel reduction at the end of the shader. Lines 13—17 implements optimized variant of parallel reduction for Nvidia HW assuming warp size is 32 threads. It changes depending on input parameters of our translator. For example, we can turn off optimization or assume smaller warp size (8 for mobile GPUs), or use *subgroupAdd* instead of 13—17 lines (available only in GLSL).

## 3.2. Code generation

The proposed generator works on the principle of code morphing [47]. The essence of this approach is that, having a certain class in a program and transformation rules we can automatically generate another class with the desired properties (for example, the implementation of the algorithm on the GPU). The transformation rules are defined by mentioned patterns, within which the processing and translation of the current code is carried out. The generated class in inherited from the input class, thus having access to all data and functions of input class.

Input source code is processed via clang and libtooling [48]. Almost all tasks in our translator are done in 2 passes. On the first pass we detect patterns and their parts via libtooling: nested loops inside kernel functions, reduction access, access to class data members and et c. On the second pass we rewrite the source code using clang AST Visitors. The final source code is produced via templated text rendering approach [50]. Thus, our solution is implemented via pure source-to-source transformations and unlike Circle, for example, we don't work with the LLVM code representation. While this approach has certain limitations (we can't change input programming language, for example, to Rust or Ada which is easily achieved in the LLVM in general), it also has significant advantages:

1. The generated source code for both shaders (OpenCL C for clspv [45] or GLSL) and host C++ with Vulkan calls looks like a normal code written by hand. It can be *debugged*, *changed or combined* with another code (hand written usually) in any way. Thus, unlike many existing programming technologies it is easy to distinguish errors of generator/translator from user errors. This is a problem for OptiX or Circle for example because we can't see what programming technology actually does with the input code.

2. The ability to generate source code for shaders gives us a huge flexibility by the subsequent shader compiler features because *we can easily add different HW features support*. The early version of our tool used only clspv [45] for shaders. However, we quickly found that the capabilities of the clspv are not enough for ray queries, virtual functions and many other things. It is possible to add such support to clspv to get desired features in SPIR-V from OpenCL C shader source code, but this is expensive and hard way because both working with SPIR-V and clspv source code requires special knowledge and significant effort. At the same time, adding new HW feature support directly to the generated GLSL source code is relatively easy.

**CPU <=> GPU data transfer.** As were mentioned in a related work review, many existing solutions solve the problem of data copying automatically. In most cases for software that uses Vulkan this is not satisfactory for many reasons. In the proposed approach, we generate code for executing algorithms on the GPU and performing copying and then let the user independently call this code. The generated function called "UpdateAll" performs this task. If user needs data back on the CPU from some internal data of generated class, he or she could make a new class, which is inherited from the generated one. In this class any additional algorithm or copying functionality can be implemented.

Our solution implements the entire generated algorithm to the GPU, therefore, in general, all generated variables and buffers are located on the GPU. However, since the generated class is inherited from the original one, it also contains all the original variables and vectors on the CPU under their own names. The user either provides his own copy implementation to "UpdateAll" method (via the interface implementation), or uses ours from the library. Accordingly, temporary buffers are either created manually by the user or an implementation provided by us is used to create them. In the same way use may manually clear unnecessary CPU data after UpdateAll method.

## 4. Experimental evaluation

We evaluated our approach on several applications for which we generated different implementations (GPU v1—v3) using different options of our translator (fig. 1). Unlike traditional compilers these options force our translator to apply different HW features and different actual implementations of the same algorithm on GPU. Therefore, performance difference is significant in some cases. Results of our experiments are presented in tables Table *1* and Table *2*. The GPU implementation is usually 30-100 times faster than a multicore CPU version which means performance is on desired level on average. Table 2, on the other hand, demonstrates the high labor intensity of implementing such experiments manually in Vulkan. Thus, performance study using our solution becomes easier.

**Reduction samples (#1--#3).** Here we demonstrate the ability to detect and generate different implementations of parallel reduction. Although the speedup is not very significant (which is expected on such tasks), it was stable, in contrast to the multithreaded execution on the CPU for which #1 and #2 in average were slower than single threaded (we took the smallest time over several runs). *GPU v1* is a cross-platform implementation which uses Vulkan 1.0, doesn't know warp size and doesn't use subgroup operations. *GPU v2* knows warp size (passed via command line argument) and thus may omit synchronization operations for several last steps of reduction (listing 3). *GPU v3* knows warp size and additionally uses subgroup operations.

**Figure 1:** Example applications of the proposed technology. Bloom filter (top, left), spherical harmonics integration (top middle), guided Non-Local Means denoising (bottom left and bottom middle), path tracing with different materials (for testing virtual functions) and procedural textures (right) and finally NBody simulation is on the top-right corner of the image.

**Table 1**

Execution time in milliseconds for different algorithms and different generated implementations via proposed technology. Because applications are different, v1—v3 means different optimizations for different samples. It is described in details further. First two rows show time in milliseconds for calculating the sum of 1 million numbers. This task is mapped to parallel reduction on GPU. The third row is spherical harmonic evaluation which is 2D reduction with some math. NLM means guided Non-Local Means filter. For path tracing implementation on the CPU we used Embree ray tracing. CPU used is Intel Core i9 10940X, GPU is Nvidia RTX 2080. For Path tracing 512x512 image was rendered (i.e., 256K paths were traced). '*' means that for path tracing and v3 variant the generated code was finalized by hand due to early stage of GLSL generator in our solution.

| App/<br>Impl | CPU (1 core)<br>Input source | CPU (14 cores)<br>OpenMP | GPU v1<br>Ours | GPU v2<br>Ours | GPU v3<br>Ours |
|---|---|---|---|---|---|
| (#1) Int Arr. Σ | 1.263 ms | 0.271 ms | 0.095 ms | 0.089 ms | **0.084 ms** |
| (#2) Float Arr. Σ | 1.420 ms | 0.342 ms | 0.104 ms | 0.096 ms | **0.096 ms** |
| (#3) Sph. Eval. | 39.73 ms | 2.931 ms | 0.399 ms | 0.364 ms | **0.320 ms** |
| (#4) NBody | 250400 ms | 11920 ms | **118.0 ms** | - | - |
| (#5) Bloom | 711.8 ms | 52.74 ms | **0.733 ms** | 1.420 ms | 0.841 ms |
| (#6) NLM | 88440 ms | 6851 ms | 422.0 ms | 571.1 ms | **351.0 ms** |
| (#7) Path Trace | 188.4 ms | 14.928 ms | 4.790 ms | 1.310 ms | **0.460 ms*** |

**Nbody (#4)** is classic GPGPU problem of a quadratic complexity for which we demonstrate 100 times acceleration in comparison to multi-threaded CPU version.

**Bloom and Non-Local Means (#5, #6).** In these samples we demonstrate the ability to change implementation of images from buffers *(GPU v1)* to textures *(GPU v2)* and use half float for texture format *(GPU v3)* which gives essential speed-up. It is interesting to note that for these image processing examples 32-bit float textures were slower than buffer variants. For Bloom buffers were even faster than a half-float textures which is due to the Load/Store access and general texture layout. In this way we show that performance question on GPU is not trivial and require to implement and test different variants of same algorithm. Withing proposed solution these experiments can be automated.

**Table 2**

Lines of code for different application. The first (C++) column shows original lines count for input class. The second column shows total line number for generated source code. Vulkan (compact)

approximately estimates number of lines for the Vulkan code written by human without our approach. We did this by excluding descriptor sets setup from the generated code because it is quite verbose in our generator and it can be written in a more compact way manually. The last column shows lines number for device code in generated kernels.

| App/Lines | C++ (input source) | Vulkan (generated) | Vulkan (compact) | Vulkan (shaders only) |
|---|---|---|---|---|
| (#1) Int Arr. Σ | 80 | 640 | 500 | 195 |
| (#2) Float Arr. Σ | 80 | 780 | 650 | 285 |
| (#3) Sph. Eval. | 120 | 1280 | 1000 | 445 |
| (#4) NBody | 115 | 850 | 700 | 140 |
| (#5) Bloom | 300 | 1460 | 1050 | 250 |
| (#6) NLM | 330 | 1290 | 1000 | 250 |
| (#7) Path Trace | 800 | 4500 | 3200 | 1470 |

For Bloom we get 100x times speedup over multi-threaded CPU version and for Non-Local Means it is only 20x, which seems to be suboptimal for such a heavy task. In fact, there could be a lot of optimizations for image processing (at least more aggressive pixel quantization/compression), but we believe Halide [35-37] project did most of them and Taiichi did similar for physics simulation [53]. So, we decided to focus our performance investigation on ray tracing. For path tracing the initial generated code (which uses exactly same traversal algorithm) outperforms original CPU variant at factor of 10x. We then replace CPU ray traversal with optimized Embree (table 2) and got only 3x. Then we add hardware accelerated ray tracing in computer shader (12x) and generate single kernel via RTX pipeline which finally gives us 32x over multithreaded CPU path tracing with Embree.

## 4.1. Path Tracing experiments (#7)

Light transport simulation algorithms involve heavy mathematical models and require extensibility from the framework in which these algorithms are implemented. In existing CPU rendering systems, this usually means object-oriented approach. Therefore, it is not yet enough to accelerate ray tracing. To achieve efficiency on GPU we should study how complex and extendible code can be implemented on GPU. Currently, there are three general approaches:

1. Single-kernel – the whole code for light path evaluation is placed inside a single kernel. There could be different option for optimizations (like OptiX state machines) [50]. This approach is good for relatively simple models, for example in computer games. However, with the growing code base it's performance dramatically reduces due to branch divergence and register pressure.
2. Multiple-kernel – code is split into several smaller kernels, which communicate by reading/writing data into GPU memory. The necessity to read/write data from/into memory results in significant performance overhead depending on an application [51]
3. Wavefront path tracing. This approach extensively uses sorting and compaction of threads to organize them into several queues which execute different computational kernels. This helps to avoid branch divergence but may result in even bigger performance overhead [52].

Therefore, all of these approaches can be used (and are used) in real life applications and there is no single approach that is strictly better than the others in all cases. Depending on the available hardware and needed features, different implementations may be needed to achieve optimal performance. Even though the implementation in code of all these approaches is significantly different, the essential algorithm (path tracing) and implemented models (BRDFs) stay the same. The actual difference in these approaches is actually how these computer graphics algorithms are translated to the GPU code.

## 4.2. Adding hardware ray tracing

We implemented the basic path tracing algorithm and several material models on the CPU and used the proposed solution to generate the GPU Vulkan-based implementation with software ray tracing in

a multiple-kernel way *(GPU v1)*. The generation was performed using ray tracing pattern previously described in section 3.1. Generated GPU version corresponds to the CPU version and implements naive path tracing algorithm consisting of the following kernel launches:

1. Primary ("eye") ray generation.
2. Loop until maximum tracing depth is reached:
    a. Ray trace kernel, which searches for intersection and stores surface hit (6 floats).
    b. Kernel to obtain material Id for hit surface (store 2 floats).
    c. Next bounce kernel which performs shading computations and stores the path state: new ray position and direction (8 floats) and accumulated color and throughput (8 floats).
3. Kernel which contributes accumulated color to the output image.

Therefore, the total size of ray pay-load is equal to 24 floats (96 bytes) per thread.

Next, the host part of generated code was modified via virtual function override (we override generated ray tracing kernel call) to use the hardware accelerated ray tracing feature via *VK_KHR_ray_query* extension which allows using RTX functionality in the compute shaders (among others). Modification required less than 1000 lines of code (about the same as drawing a single triangle in Vulkan). This is a *GPU v2*. In way we show how generated and hand written code could be connected together (both CPU and GPU parts). Considering kernel launches described above, we simply replace Ray trace kernel with the new one which uses hardware accelerated ray queries.

Finally, we have implemented several variants of exactly the same path tracing setup via full RTX pipeline for performance comparison. We took a part of generated GLSL code (functions of material sampling, etc.) and finalized it by adding to ray tracing pipeline. This is a *GPU v3*. In fact, there were 3 different implementations of *GPU v3* because RTX itself has many options (fig. 2, first 3 rows).

In all cases path tracing was performed with tracing depth equal to 6 and with 8 samples per pixel (for larger sample numbers Nvidia Nsight runs out of memory for GPU trace). Measurements were made on a geometrically simple scene (about 31k triangles), but featuring a variety of material models - Lambertian, perfect mirror, glossy GGX and a blended material – GGX and Lambertian BRDF mixed with respect to a procedural noise texture mask.

Initially we didn't plan to generate single kernel version because for offline ray tracing applications it's not the best option due to significant performance degradation when adding new materials and light source models. We didn't even plan to generate GLSL code using our generator for logic, opposite we plan to replace specific parts of the algorithm (for example, ray tracing) via separate kernel calls. Interaction via DRAM seems to be natural and common for GPU programs, but it turned out that this approach is rather limited. First, clspv has only basic support for hardware features in shaders. Second, for relatively simple code base single kernel variant can be significantly better because less data is transferred to DRAM from chip (fig. 2). Nevertheless, our *GPU v2* implementation almost caught up with the most flexible/stable variant of RTX via callable shaders (first row in fig. 2). This one seems to be a wavefront path tracing approach implemented by Nvidia inside RTX pipeline and it's not cross-platform. Thus, our further studies were related to a question: can we get the same performance as a solution based on callable shaders within the multiple-kernels framework by, for example, regrouping threads on material sampling to avoid branch divergence and high register pressure.

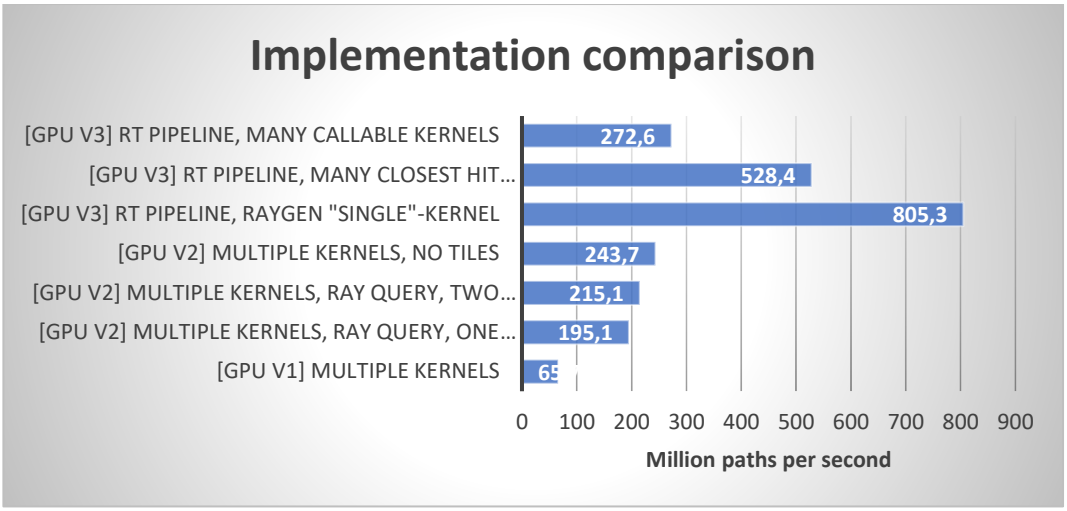


**Figure 2:** Comparison of performance in millions of paths per second between different variants; 1024x1024; RTX2080. Raygen "single"-kernel variant implements all material models inside ray generation shader, many closest hit kernels variants perform computations for different material models inside different closest hit shaders and many callable kernels – inside different callable shaders invoked from ray generation shader. The last 4 rows is GPU v2 (ray query, with different tailing variants) and GPU v1.

## 4.3. Performance analysis and asynchronous compute for multiple kernel

A problem of multiple kernel approach is that data which passes between kernels is stored in DRAM. Actually, if the size of the intermediate data is not large and it can fit into L2 cache, the work load of DRAM significantly decreases which we analyzed via Nvidia Nsight tool. Unfortunately, we cannot significantly reduce the number of active threads because barriers between kernels lead to a frequent situation when a previous kernel is still computing in a few threads, but the new one can't be launched. So, for arbitrary tile size there is a tradeoff between L2 hit rate and amount of required memory and DRAM throughput for multiple kernel approach (fig. 3, blue lines).

In fact, for multiple kernel approach we should try to reduce tile size because less memory will be required for intermediate data and DRAM workload will be reduced. This usually means more complex stuff can be done efficiently in future. To do this efficiently we have to get new *independent* work to GPU as previous kernel finishes execution. Thus, we decided to submit new work from independent queue using *asynchronous compute* in Vulkan (fig. 3, orange lines). Let's say we have 1024x1024 image and we have split it into tiles with the size of 256x256 pixels. We can process the whole image tile by tile, or we can, for example, process two 256x256 tiles asynchronously. For fair comparison we should take tile size twice as big for single queue as it is for two queues (for example, 512x256 for a single queue and 256x256 for two queues) so the actual buffer size would be the same. Even then, having two asynchronous queues shows ~10-12% better performance in the best cases on Nvidia GPU and up to 16% on AMD (fig. 3) over tile-by-tile approach. It can be seen from fig. 4 and 5 that AMD hardware implements asynchronous compute significantly better than Nvidia.

Asynchronous tile-based approach implementation does not require many modifications to the code – we need to create additional queue from a different queue family, record command buffers for each tile using alternating queues (each tile launches the same kernels as described in 4.2) and submit the commands in a multithreaded fashion (so we won't block on fence synchronization on the CPU). Note that the number of executed command buffers depends linearly on the number of tiles.

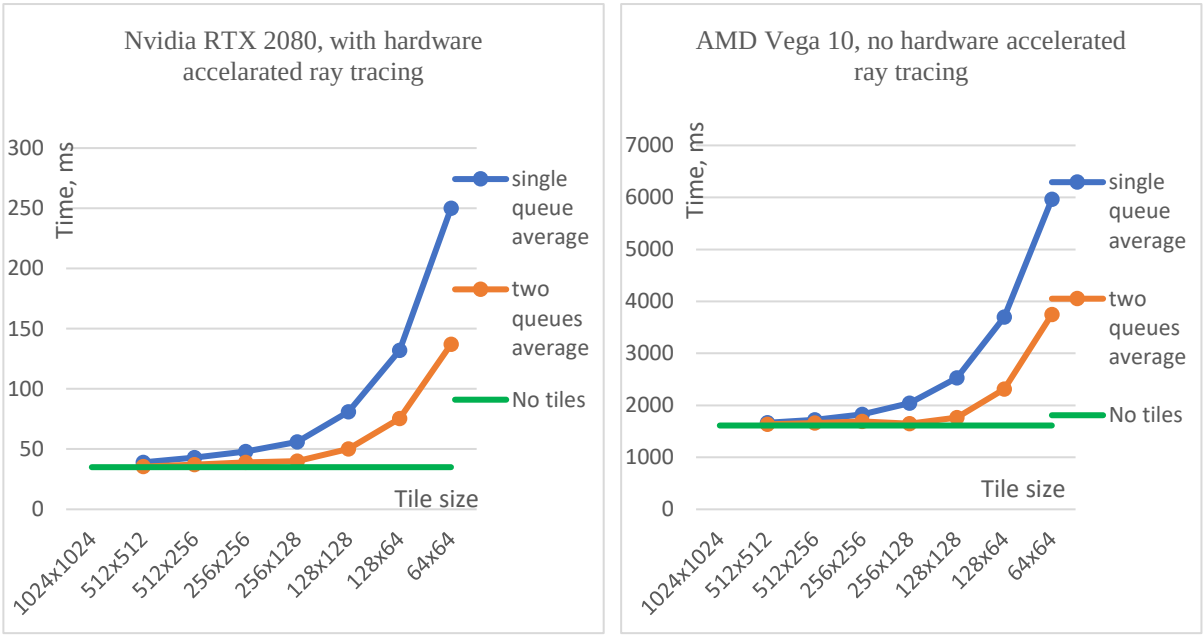


**Figure 3:** Measurements for (left) RTX 2080 with hardware accelerated ray tracing (*VK_KHR_ray_query, GPU v2*) and (right) AMD Vega 10 without it (GPUv1), 8 samples per pixel, ray tracing depth = 6, total image resolution 1024x1024. DRAM throughput decreases linearly from 90% (1024x1024) to 10% (for 128x128) and even less for smaller tile size. This is not shown on the image. Asynchronous compute (orange lines) shows significant performance increase over simple multiple kernel approach.

Although rendering without splitting the image into tiles for this simple scene is still slightly faster, the difference is not essential (1-2%) and we've got significantly better HW metrics for units for proposed tiled rendering (table 3). So, with more complex materials tile splitting may actually become a better option. At the same time our approach reduces memory required for intermediate data up to 8-16 times, depending on the tile size. This can be especially important for MCMC methods (Metropolis Light Transport) where large vectors are stored for each thread.

**Table 3**

Nvidia RTX 2080, Nsight Graphics Metrics

| Metric | No tiles | One 512x256 Tile | Two 256x256 tiles |
|---|---|---|---|
| **VRAM throughput** | 48.9% | 35.3% | 23.8% |
| **L2 hit rate** | 23.1% | 28.9% | 48.7% |
| **L2 hit rate from L1** | 21.7% | 28.4% | 47.9% |
| **CS Warp can't launch (register limited)** | 31.6% | 15.4% | 6.3% |
| **Average time** | 35 ms | 43 ms | 39 ms |

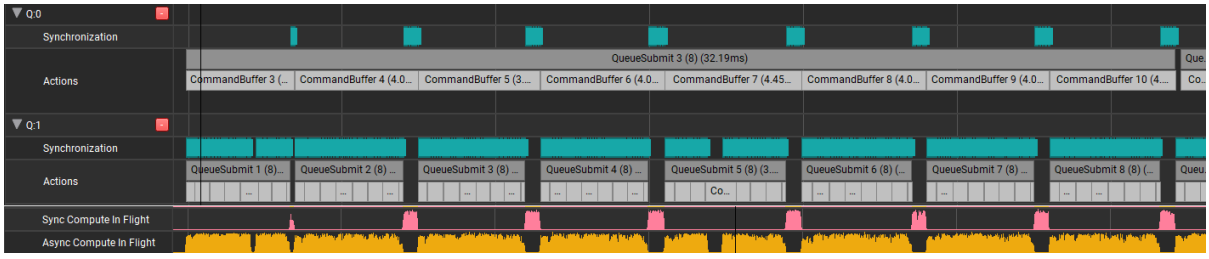


**Figure 4:** Nsight Graphics GPU trace for RTX2080. Path tracing with asynchronous compute queues from different queue families. An asynchronous compute queue on Nvidia (top part of the image) runs like a "background" task. It can be seen that 8 submits from the main queue (bottom part of the image) takes same time than single submit to async compute queue.

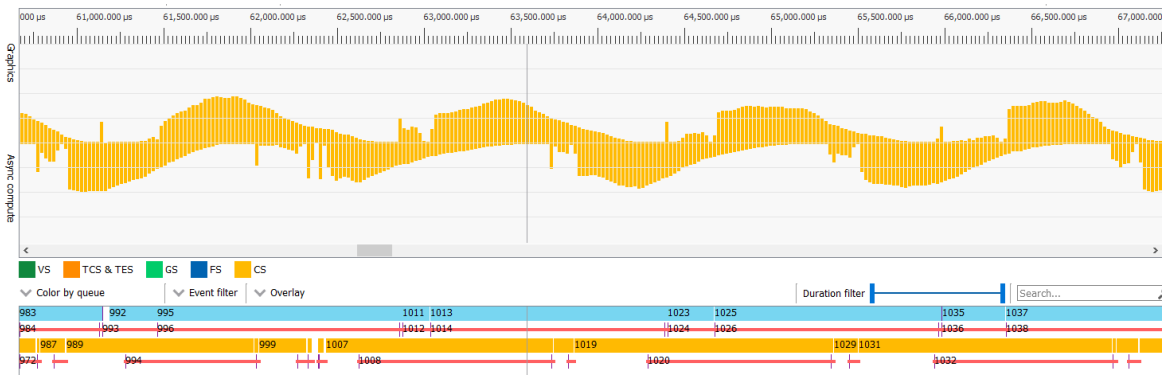


**Figure 5:** Radeon GPU Profiler frame capture for Vega 10. Path tracing with asynchronous compute queues from different queue families. Unlike Nvidia, the workload floats freely between 2 queues.

## 5. Conclusions

We proposed a solution capable of alleviating the difficulties of porting a computationally intensive algorithm to the GPU using source-to-source translation of ordinary C++ code to C++ with necessary Vulkan API calls and shader code (OpenCL C or GLSL). During the code generation process different optimizations can be applied to create several implementations depending on the problem specifics and/or available hardware. This way we are able to increase GPU developers' productivity by generating code using Vulkan (which can be quite verbose) and applying complex optimizations automatically to achieve maximum performance. We have shown that generated code could be connected to hand written adding hardware accelerated ray tracing. Finally, using the proposed solution and asynchronous compute in Vulkan we made a performance study for path tracing via multiple kernel approach and proposed an improvement for it which reduces required memory for intermediate data by an order of magnitude and uses GPU memory units in a more efficient way (table 3).

## 6. Acknowledgments

This work is supported by the Russian Science Foundation (RSF) under grant #21-71-00037.

## 7. References

[1] J. Fang, et al., Parallel programming models for heterogeneous many-cores: a comprehensive survey, CCF Transactions on High Performance Computing 2(4) (2020) 382-400.

[2] Thrust: a powerful library of parallel algorithms and data structures, 2021. URL: https://developer.nvidia.com/thrust

[3] A. Kolesnichenko, C. M. Poskitt, S. Nanz, SafeGPU: Contract-and library-based GPGPU for object-oriented languages, Computer Languages, Systems & Structures 48 (2017) 68-88.

[4] D. Beckingsale, et al., Performance portable C++ programming with RAJA. In: Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PpoPP, 2019.

[5] T. Heller, et al., Hpx–an open source c++ standard library for parallelism and concurrency, in: Proceedings of OpenSuCo, 2017, 5.

[6] A. Paszke, et al., Pytorch: An imperative style, high-performance deep learning library, arXiv preprint (2019) arXiv:1912.01703.

[7] M. Abadi, et al., Tensorflow: Large-scale machine learning on heterogeneous distributed systems, arXiv preprint (2016) arXiv:1603.04467.

[8] T. Chen, et al., Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, arXiv preprint (2015) arXiv:1512.01274.

[9] OpenACC, 2021. URL: https://www.openacc.org/

[10] N. Jacobsen, LLVM supported source-to-source translation-Translation from annotated C/C++ to CUDA C/C++, University of Oslo, Norway, 2016.
[11] G. D. Balogh, et al., Op2-clang: A source-to-source translator using clang/llvm libtooling, in: Procceedings of IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), 2018.
[12] P. Yang, et al., Improving utility of GPU in accelerating industrial applications with user-centered automatic code translation, IEEE Transactions on Industrial Informatics 14(4) (2017) 1347-1360.
[13] J. A. Pienaar, S. Chakradhar, A. Raghunathan, Automatic generation of software pipelines for heterogeneous parallel systems, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12, IEEE, 2012.
[14] N. Konovalov, V. Kryakov, 2002. URL: https://www.keldysh.ru/dvm/dvmhtm1107/publishr/dvm-appr-OpSys.htm (in Russian).
[15] V. Bakhtin, et al., 2008. URL: http://agora.guru.ru/abrau2008/pdf/085.pdf (in Russian).
[16] V.A. Bakhtin, V.A. Krukov, DVM-Approach to the Automation of the Development of Parallel Programs for Clusters, Programming and Computer Software, 3 (2019) 43-56.
[17] M. A. Mikalsen, Openacc-based snow simulation, MS thesis. Institutt for datateknikk og informasjonsvitenskap, 2013.
[18] T. Öhberg, Auto-tuning Hybrid CPU-GPU Execution of Algorithmic Skeletons in SkePU, MS thesis. Linkoping University, 2018.
[19] A. Ernstsson, L. Lu, C. Kessler, SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems, International Journal of Parallel Programming 46(1) (2018) 62-80.
[20] M. Steuwer, P. Kegel, S. Gorlatch, Skelcl-a portable skeleton library for high-level gpu programming, in: Proceedings of 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, IEEE, 2011.
[21] SYCL, cross-platform abstraction layer, 2021. URL: https://www.khronos.org/sycl/
[22] SYCL for CUDA developers, examples, 2020. URL: https://developer.codeplay.com/products/computecpp/ce/guides/sycl-for-cuda-developers/examples
[23] T. D. Han, T. S. Abdelrahman, hiCUDA: High-level GPGPU programming, IEEE Transactions on Parallel and Distributed systems 22(1) (2010) 78-90.
[24] J. Wu, et al., gpucc: an open-source GPGPU compiler, in: Proceedings of the 2016 International Symposium on Code Generation and Optimization, 2016.
[25] P. Sathre, M. Gardner, W. Feng, On the portability of cpu-accelerated applications via automated source-to-source translation, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, 2019.
[26] HIP, C++ Runtime API and Kernel Language, 2021. URL: https://github.com/ROCm-Developer-Tools/HIP
[27] Vulkan specification, indirect dispatch command, 2021. URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/vkCmdDispatchIndirect.html
[28] Anton Sherin. Resident Evil 2 Frame Breakdown, 2019. URL: https://aschrein.github.io/2019/08/01/re2_breakdown.html
[29] N. Mammeri, B. Juurlink, Vcomputebench: A vulkan benchmark suite for gpgpu on mobile and embedded gpus, in: Proceedings of IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2018.
[30] Nvidia OptiX, 2021. URL: https://developer.nvidia.com/optix
[31] DIrectML, 2021. URL: https://github.com/microsoft/DirectML
[32] Wisp renderer, 2021. URL: https://github.com/TeamWisp/WispRenderer
[33] Projects using RTX, 2021. URL: https://github.com.cnpmjs.org/vinjn/awesome-rtx
[34] J. Hegarty, et al., Darkroom: compiling high-level image processing code into hardware pipelines, ACM Trans. on Graphics 33(4) (2014) 144-1.
[35] J. Ragan-Kelley, et al., Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM Sigplan Notices 48(6) (2013) 519-530.

[36] R. T. Mullapudi, et al., Automatically scheduling halide image processing pipelines, ACM Transactions on Graphics (TOG) 35(4) (2016) 1-11.
[37] A. Adams, et al., Learning to optimize halide with tree search and random programs, ACM Transactions on Graphics (TOG) 38(4) (2019) 1-12.
[38] A. Rasch, R. Schulze, S. Gorlatch, Developing High-Performance, Portable OpenCL Code via Multi-Dimensional Homomorphisms, in: Proceedings of the International Workshop on OpenCL, 2019.
[39] T.-W. Huang, et al., Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2021).
[40] M. Haidl, S. Gorlatch, PACXX: Towards a unified programming model for programming accelerators using C++14, in: Proceedings of Workshop on the LLVM Compiler Infrastructure in HPC, IEEE, 2014.
[41] M. Haidl, et al., Pacxxv2+ RV: an LLVM-based portable high-performance programming model, in: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, 2017.
[42] A. Sidelnik, et al., Performance portability with the chapel language, in: Proceedings of the IEEE 26th international parallel and distributed processing symposium, 2012.
[43] Y. Hu, et al., Taichi: a language for high-performance computation on spatially sparse data structures, ACM Transactions on Graphics (TOG) 38(6) (2019) 1-16.
[44] R. Baghdadi, et al., Tiramisu: A polyhedral compiler for expressing fast and portable code, in: Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2019.
[45] Google clspv. A prototype compiler for a subset of OpenCL C to Vulkan compute shaders, 2021. URL: https://github.com/google/clspv
[46] Sean Baxter. Circle C++ shaders, 2021. URL: https://github.com/seanbaxter/shaders
[47] S. S. Huang, D. Zook, Y. Smaragdakis, Morphing: Safely shaping a class in the image of others, in: Proceedings of European Conference on Object-Oriented Programming, Springer, Berlin, Heidelberg, 2007.
[48] Clang documentation, 2021. URL: https://clang.llvm.org/docs/LibTooling.html
[49] Inja, template engine for modern C++, 2021. URL: https://github.com/pantor/inja
[50] S. G. Parker, et al., Optix: a general purpose ray tracing engine, ACM transactions on graphics 29(4) (2010): 1-13.
[51] V. A. Frolov, V. A. Galaktionov, Memory-compact Metropolis light transport on GPUs. Program. Comput. Software 43 (2017) 196–203. doi:10.1134/S0361768817030057
[52] S. Laine, T. Karras, T. Aila, Megakernels considered harmful: Wavefront path tracing on GPUs, in: Proceedings of the 5th High-Performance Graphics Conference, 2013.
[53] Y. Hu, J. Liu, X. Yang, M. Xu, Y. Kuang, W. Xu, Q. Dai, W. T. Freeman, F. Durand, QuanTaichi: A Compiler for Quantized Simulations, ACM Transactions on Graphics 40(4) (2021).