High-level Programming of Vulkan-based GPUs through OpenMP

Ilias K. Kasmeridis and Vassilios V. Dimakopoulos

HLPP 2025
18th International Symposium on
High-level Parallel Programming and Applications

Department of
Computer Science & Engineering

University of Ioannina

PARALLEL
PROCESSING
GROUP

Outline

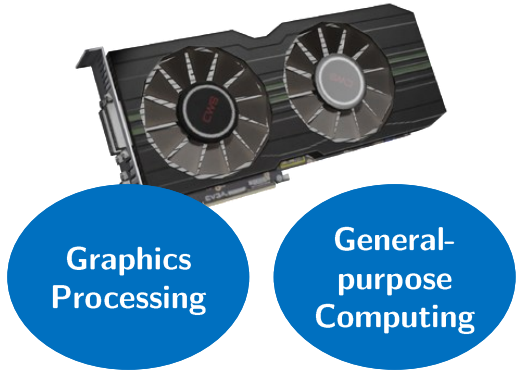
- Introduction
- Translating OpenMP to Vulkan shaders
- Vulkan offloading plugin
- Evaluation
- Conclusion & future work

Introduction

Background

- Recently, heterogeneous systems have become widely adopted
  - Increasing demand for high-performance computing and graphics

- Various types of accelerators incorporated
  - GPUs — most popular
  - FPGAs
  - Co-processors



Background (cont.)

- Programming GPUs traditionally required low-level API usage
  - CUDA
  - OpenCL
  - Vulkan Compute (more recent)

- Multiple challenges
  - Different codebases required (code heterogeneity)
  - Steeper learning curve
  - Large amounts of device-specific code
    - Initialization, finalization, data transfers, kernel offloading…
  - Difficult code maintenance

- Increased demand for higher-level programming models

Embedded GPUs

- Heterogeneous computing evolves
  - GPU architectures are diversifying
  - Embedded, low-power GPUs gain popularity
- Support for such GPUs is not guaranteed by dominant APIs
  - CUDA is limited to NVIDIA GPUs
  - Alternatives like OpenCL not always available

OpenMP

- OpenMP is a de-facto higher-level programming model
- Version 4.0 introduced the device model
  - Allows parts of a program to be offloaded to installed devices (e.g., GPUs)
- Computations/data offloaded from CPU to connected devices through constructs
  - `#pragma omp target […]`

Vulkan Compute

- Vulkan Compute
  - Lower-level than the dominant APIs…
    - Compute pipeline & buffer setup left to the programmer
  - …but can target a wider variety of GPUs!
    - From embedded/specialized (e.g. Raspberry Pi VideoCore)
    - To discrete, off-the-shelf ones (e.g. NVIDIA & AMD GPUs)
  - Suitable for use as a back-end by high-level programming models

- Idea: use Vulkan Compute as an OpenMP offloading target
  - Details are abstracted away by the OpenMP device model
    - Buffer allocation, memory transfers, pipeline setup etc.
  - Performance ⬆
  - Portability ⬆
  - Ease-of-programming ⬆

Our contributions

- Our work is based on an open source OpenMP C compilation framework, OMPi
  - https://paragroup.cse.uoi.gr/wpsite/software/ompi

- Contributions:
  - A translator that produces Vulkan shaders from code associated with OpenMP `target` constructs
  - A novel runtime system that supports OpenMP on Vulkan-enabled GPUs
    - OpenMP is also supported within the offloaded code
  - An open-source implementation that is the first to provide offloading capabilities for Raspberry Pi boards and their VideoCore GPUs
  - Experimental results of our implementation on various GPUs

Translating OpenMP to Vulkan shaders

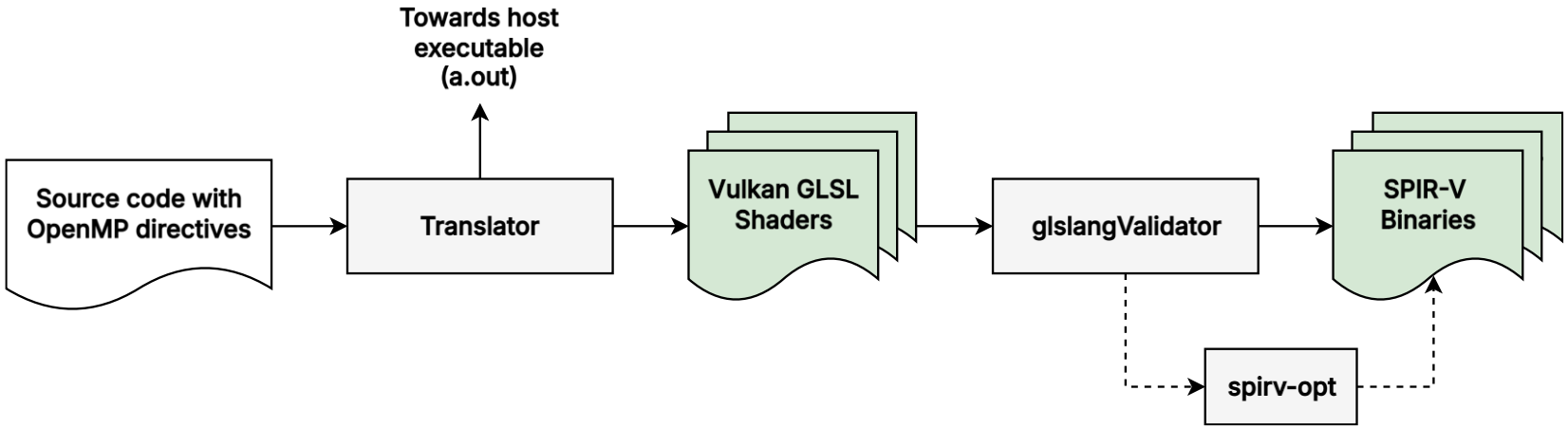
Abstract Syntax Tree

- OMPi relies on an Abstract Syntax Tree (AST) representation
  - Parsed representation of the source code
  - OpenMP target constructs are represented as nodes
- When encountering a `target` construct, OMPi generates:
  - Host CPU code that interacts with the device
    - Device management and code/data offloading
    - Standard for all device classes
  - Kernel code (extracted from the construct)
    - Offloaded to the device
    - Output to separate files, or embedded within the application

Shader files

- We implemented a converter of OpenMP C to Vulkan GLSL shaders
  - Extension to the code generation part of the compiler
- Shader files are ultimately compiled to SPIR-V binaries (.spv)
  - `glslangValidator` tool
  - Optimized with `spirv-opt` in terms of
    - Size
    - Performance
  - Intermediate .comp files are kept for manual tuning

Translation process



Outlining

- The translator relies on *outlining* to create shader files
  - Process of placing a code block *B* to a new, separated function *F*
  - In its place, the compiler inserts replacement code *R*
    - At minimum consists of a call to *F*
    - Glue code is placed before/after the call
- Analysis of variables declared in *B* is required
  - Variables must be also declared within *F*
    - …and initialized to their correct values
  - New values must be visible after returning from *F*
- For target constructs
  - *F* is placed to a new file (shader)
  - *R* is placed to the host CPU file

Target directives

- To support Vulkan devices, our translator follows a number of steps:
  - Body translation: Handle OpenMP directives within the offloaded code
  - Data analysis: Analyze used variables and append CPU-to-device (and vice versa) mapping calls to *R*
  - Cast handling: Replace C-style casting with GLSL-style casting
    - Example: `(float) x` → `float(x)`
  - Pointer handling: Minimal handling of function parameters
    - Example: `int *x` → `inout x`
  - Math functions: Replace float-specific variants (e.g., `sqrtf`) to GLSL equivalents (e.g., `sqrt`)

Kernel generation

- The last step of our translator involves kernel file generation

- Structure of shader files
  - Library code (for supporting OpenMP)
  - Buffer declarations
  - Shader function (*F*) – e.g. `vulkan_kernel1()`
  - Main function

Parallelism

- Multiple Vulkan workgroups and invocations can be launched by combining teams, target and parallel directives
  - Example:

```
#pragma omp target teams distribute parallel for num_teams(X) num_threads(Y)
<for-loop>
```

- Exact grid and workgroup dimensions can be given using a product of three factors
  - $x * y * z$

- Our translator also handles parallelism within the offloaded code, using the if-master scheme
  - The master invocation (ID=0) executes serial regions using if statement guards
  - Other invocations only execute parallel regions

Worksharing & Synchronization

- Currently, the translator supports static scheduling for OpenMP `for` and `distribute` directives
  - Use of unified target teams distribute parallel for directive is recommended
- `single` directives are also supported
  - The master invocation calls the associated code block
- Synchronization is performed using `barrier` directives
  - Replaced with `vk_barrier()` calls
  - Rely on the GLSL native `barrier()` function

Example: translating an OpenMP target directive to Vulkan GLSL

```c
/* main.c */
void func() {
   int x[16];
   #pragma omp target map(tofrom:x[0:16]) device(1)
   {
      int y = 0;
      #pragma omp parallel num_threads(16)
        x[omp_get_thread_num()] = 5;
      y = 1;
   }
}
```

```c
/* main_host.c */
void main() {
   int x[16];
   int device_id = 1, num_variables = 1;
   void *data_env = start_dataenv(num_variables, device_id);
   map_target_var(&x, device_id);
   void *args[] = { &x };
   /* from parallel region */
   int num_teams = 1, num_threads = 16;
   offload("vulkan_kernel1", args, device_id, num_teams,
           num_threads);
   unmap_target_var(&x);
   end_dataenv(data_env);
}
```

```c
/* vk_kernel1.comp */
<shader library code>
layout(set = 0, binding = 0) buffer _shader_buffer_x {
    int x[16];
};
void vulkan_kernel1() {
    /* if-master initialization */
    int __inv_id = omp_get_thread_num(),
        __num_inv = omp_get_num_threads(),
        __inv_mask = int(__inv_id == 0);
    int y;
    if (__inv_mask == 1) /* Master invocation only */
        y = 0;
    /* #pragma omp parallel num_threads(16) */
    vk_barrier();
    if (__inv_id < 16) /* Participate if ID < 16 */
        x[omp_get_thread_num()] = 5;
    vk_barrier();
    if (__inv_mask == 1)  /* Master invocation only */
        y = 1;
}

void main() {
    vulkan_kernel1();
}
```

Vulkan offloading plugin

Vulkan offloading plugin

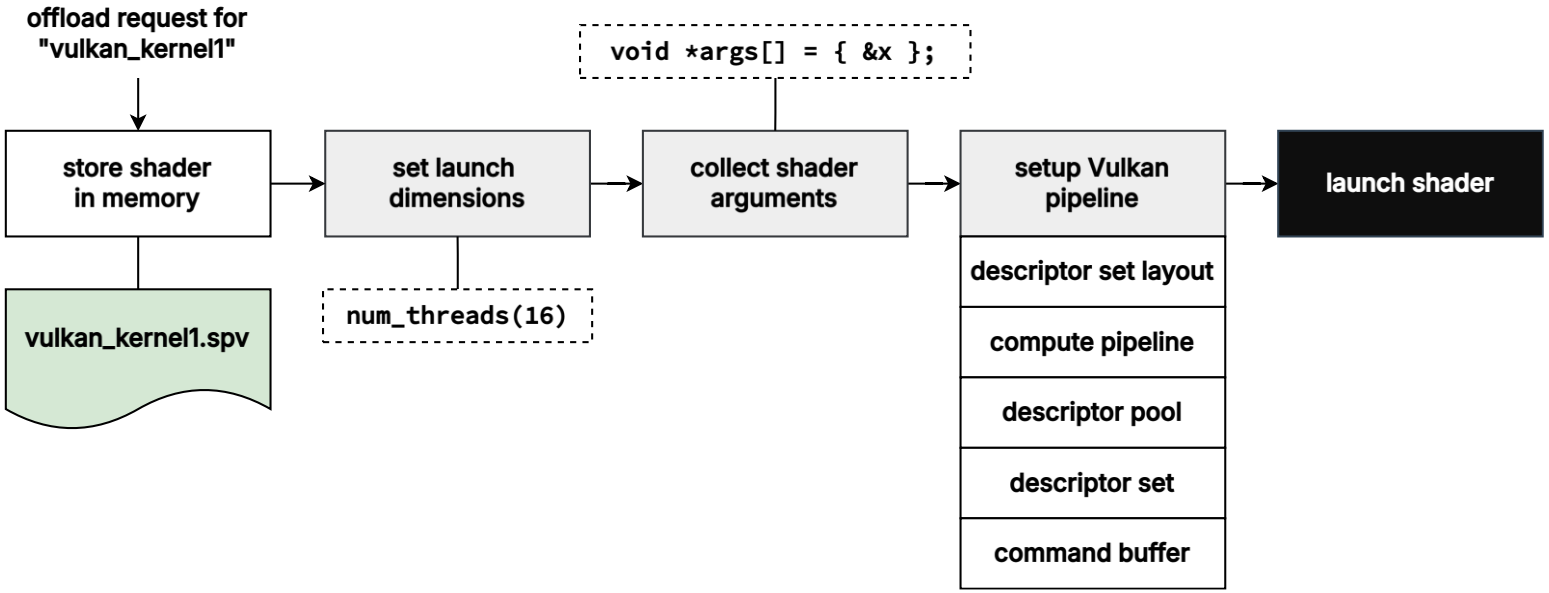
- To support OpenMP offloading to Vulkan-enabled GPUs, we implemented a dedicated runtime plugin
  - Vulkan offloading plugin
- The plugin consists of two integral parts
  - Host-device interface
    - Allows host CPU to manage a Vulkan device
  - Shader library
    - Provides OpenMP facilities within the offloaded shader code

Host-device interface

- The host-device interface relies on the Vulkan Compute API to perform actions on the GPU

- Supported operations
  - Device initialization/finalization
  - Memory allocation/deallocation
  - Data transfers (read/write)
  - Kernel offloading
    - Most complicated functionality
    - Requires loading of the shader binary, setting up compute pipeline etc.

Kernel offloading



Shader library

- OpenMP within the offloaded shader is enabled by a new, dedicated shader library which provides implementations for:
  - OpenMP API routines
  - Static loop scheduling
    - Functions that assign a set of iterations to an invocation
  - Barriers & memory fences
- The shader library code is placed in the beginning of each generated shader file

Example: translating an OpenMP target directive to Vulkan GLSL

```
/* main.c */
void func() {
  int omp_get_num_threads(void)
  {
      return int(gl_WorkGroupSize.x * gl_WorkGroupSize.y
                 * gl_WorkGroupSize.z);
  }
  …
      y = 1;
   }
}
```

```
/* main_host.c */
void main() {
   int x[16];
   int device_id = 1, num_variables = 1;
   void *data_env = start_dataenv(num_variables, device_id);
   map_target_var(&x, device_id);
   void *args[] = { &x };
   /* from parallel region */
   int num_teams = 1, num_threads = 16;
   offload("vulkan_kernel1", args, device_id, num_teams,
           num_threads);
   unmap_target_var(&x);
   end_dataenv(data_env);
}
```

```
/* vk_kernel1.comp */
<shader library code>
layout(set = 0, binding = 0) buffer _shader_buffer_x {
    int x[16];
};
void vulkan_kernel1() {
    /* if-master initialization */
    int __inv_id = omp_get_thread_num(),
        __num_inv = omp_get_num_threads(),
        __inv_mask = int(__inv_id == 0);
    int y;
    if (__inv_mask == 1) /* Master invocation only */
        y = 0;
    /* #pragma omp parallel num_threads(16) */
    vk_barrier();
    if (__inv_id < 16) /* Participate if ID < 16 */
        x[omp_get_thread_num()] = 5;
    vk_barrier();
    if (__inv_mask == 1)  /* Master invocation only */
        y = 1;
}

void main() {
    vulkan_kernel1();
}
```

Evaluation

Evaluation

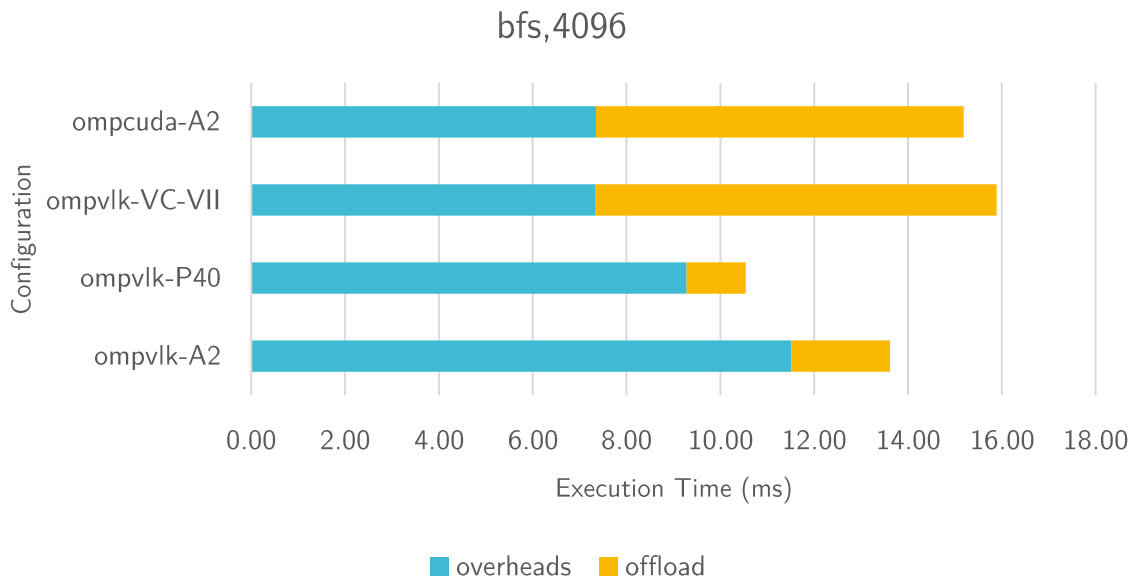
- To evaluate our implementation, we selected two benchmarks from the HeCBench suite
  - Breadth-First Search (BFS)
  - Adaptive Moment Estimation (ADAM)
- We also implemented and executed a number of image processing kernels
  - Kuwahara Filter
  - Gaussian Blur
  - Unsharp Masking
  - Sobel Filtering

System configuration

- We selected three systems for the evaluation
  - System A: Intel Xeon Gold 6130 Server (2 x CPUs, 64GB RAM)
    - 1 x NVIDIA Tesla P40 GPU
    - 3,840 CUDA cores
  - System B: AMD EPYC 9354 Server (2 x CPUs, 128GB RAM)
    - 3 x NVIDIA Ampere A2 GPUs
    - 1,280 CUDA cores/GPU
  - System C: Raspberry Pi 5 Board (8GB RAM)
    - 1 x VideoCore VII GPU
    - 8 compute units
    - 128 shader units
- Three configurations
  - `ompvlk`: Our OpenMP infrastructure
  - `ompcuda`: OpenMP Offloading to CUDA
  - `vlk`: Pure Vulkan code

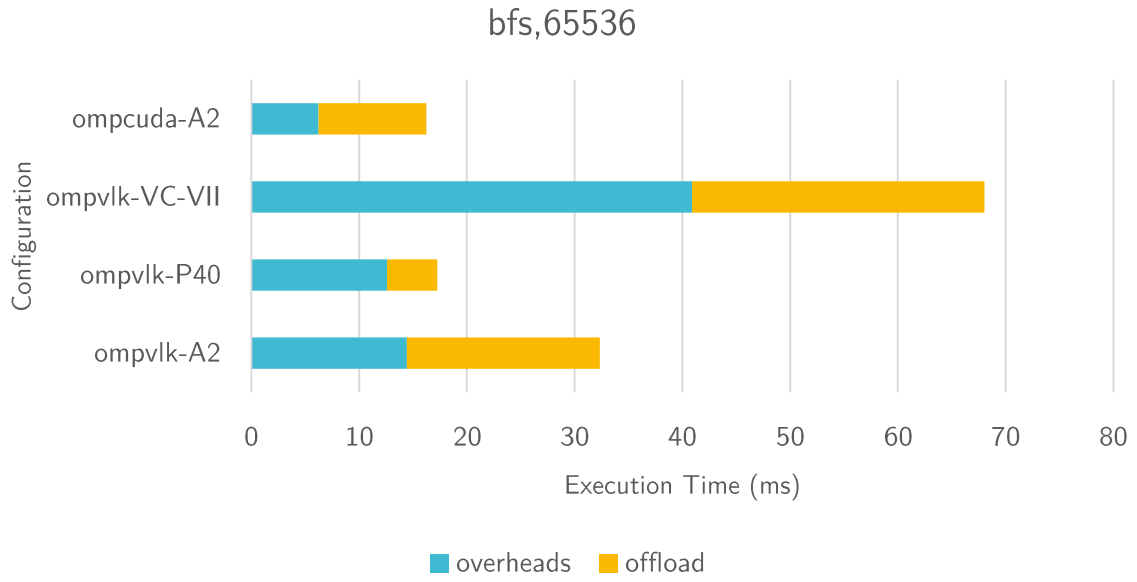
Breadth-First Search

For the Breadth-First Search (BFS) graph traversal algorithm we used two different graphs sizes (4,096 and 65,536 nodes)



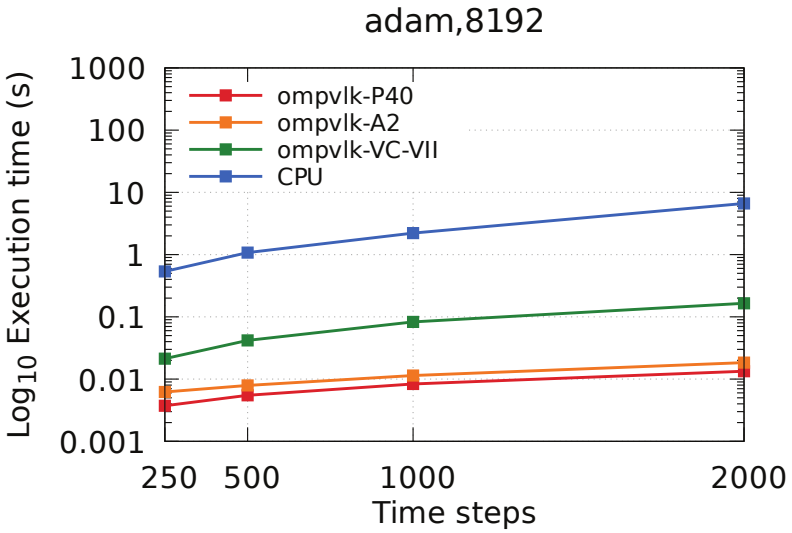
Breadth-First Search (cont.)

For the Breadth-First Search (BFS) graph traversal algorithm, we used two different graphs sizes (4,096 and 65,536 nodes)

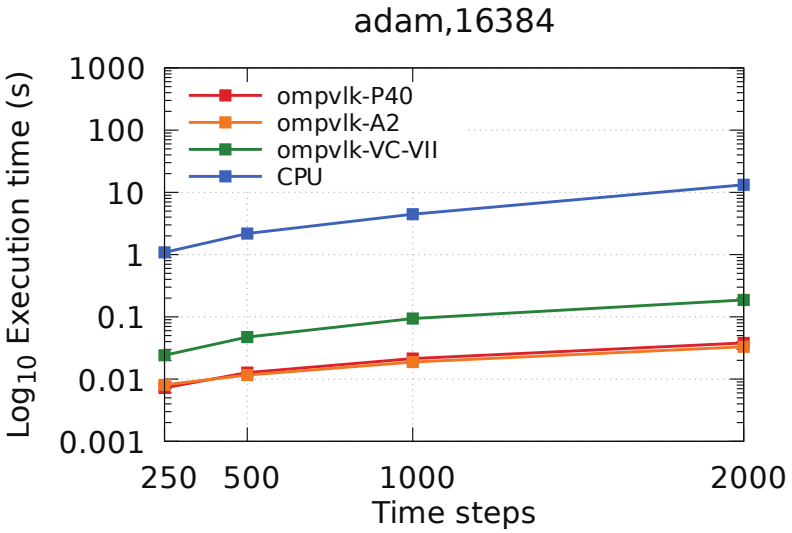


Adaptive Moment Estimation

Adaptive Moment Estimation (ADAM) is a stochastic optimization method that works on a set of parameters – we used two set sizes (8,192 and 16,384)



(a) Small parameter set

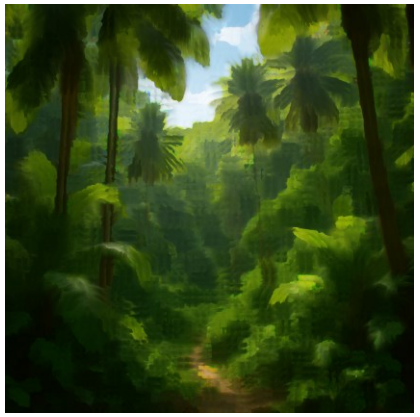


(b) Large parameter set

Image kernels

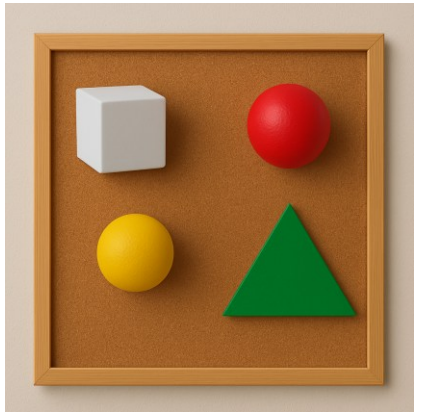
Kuwahara Filter

(a) Original

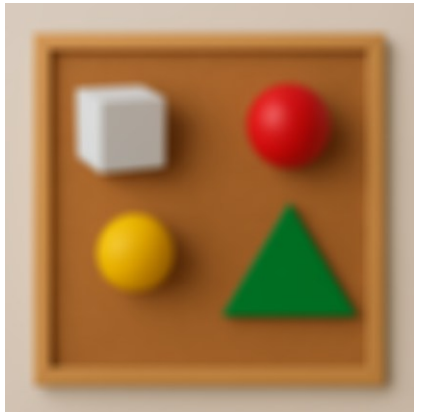


(b) R=16

Gaussian Blur



(a) Original



(b) R=16

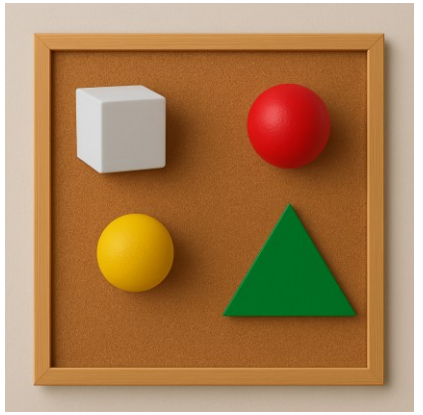
Image kernels (cont.)

Unsharp Masking

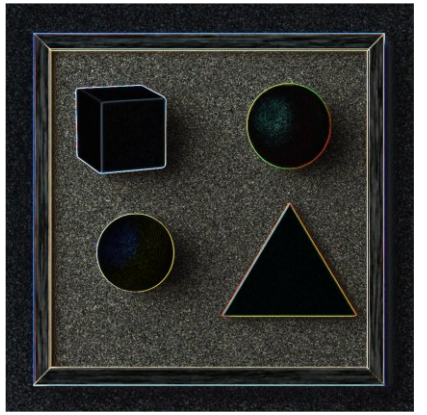
(a) Original

(b) R=16

Sobel Operator

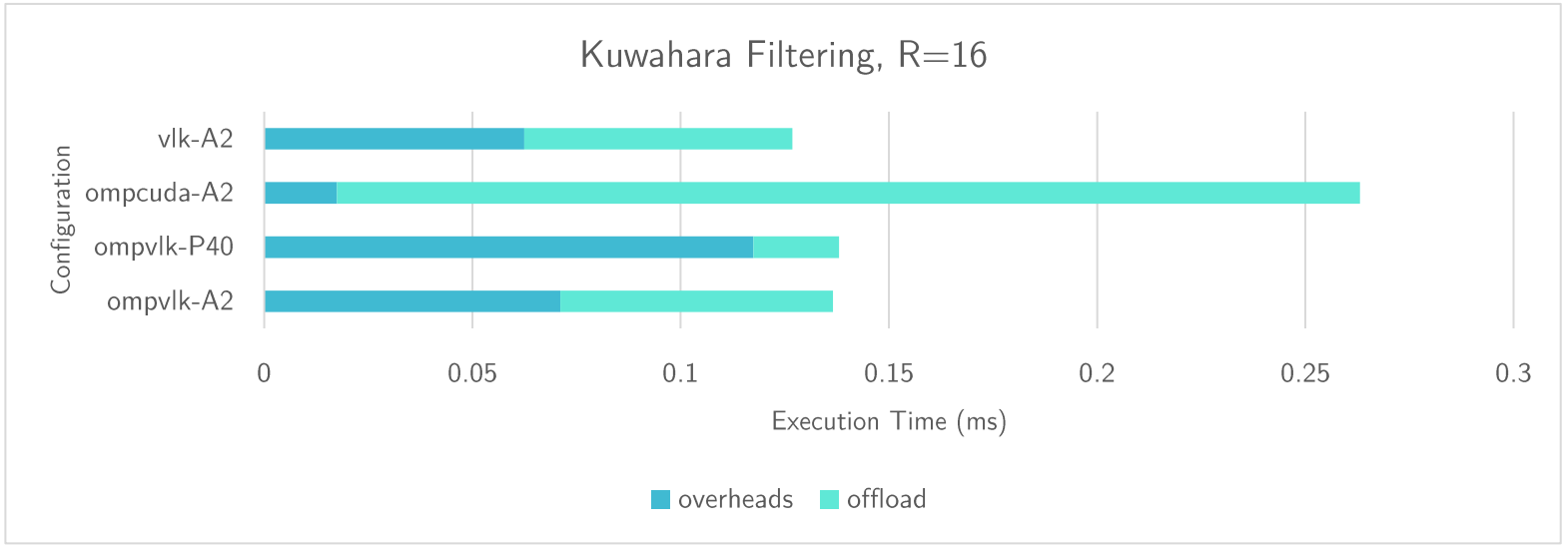


(a) Original



(b) R=1

Image kernels – Results



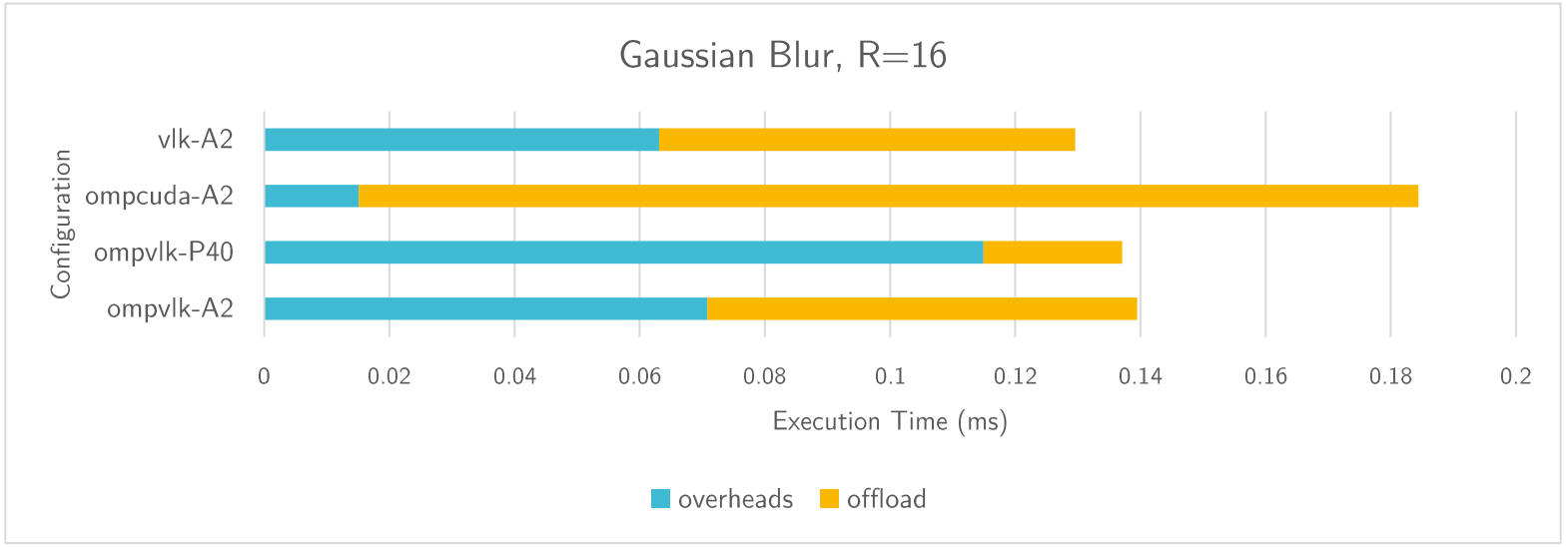
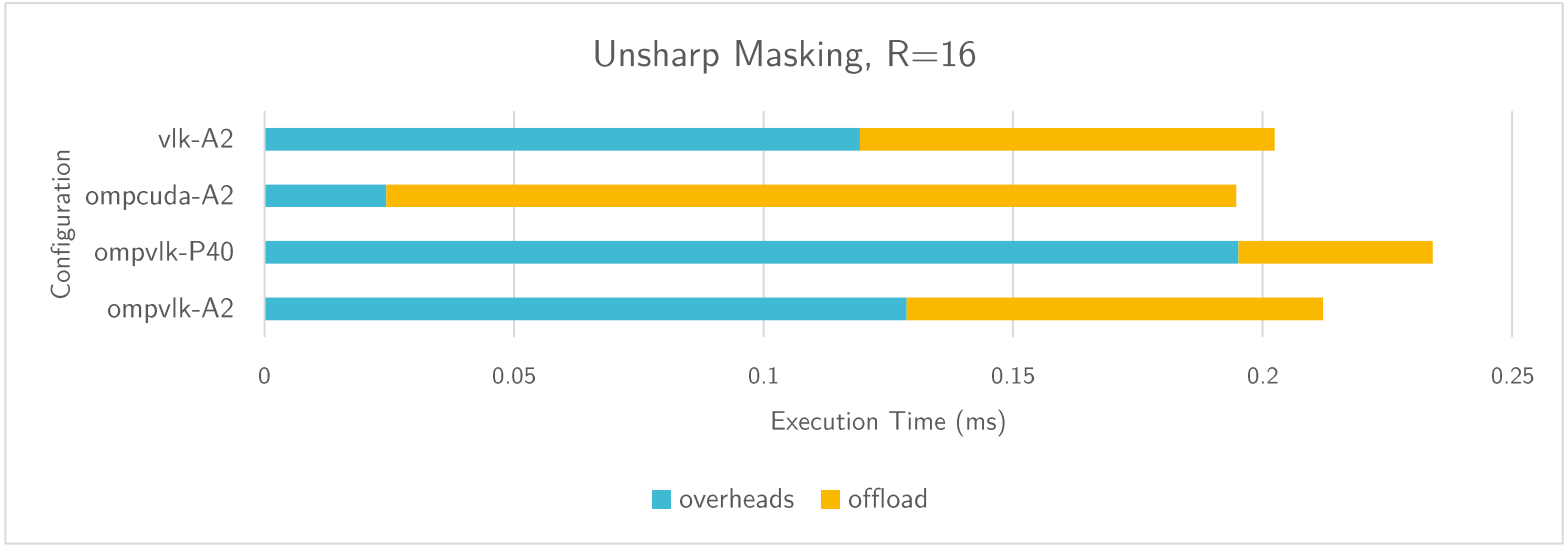
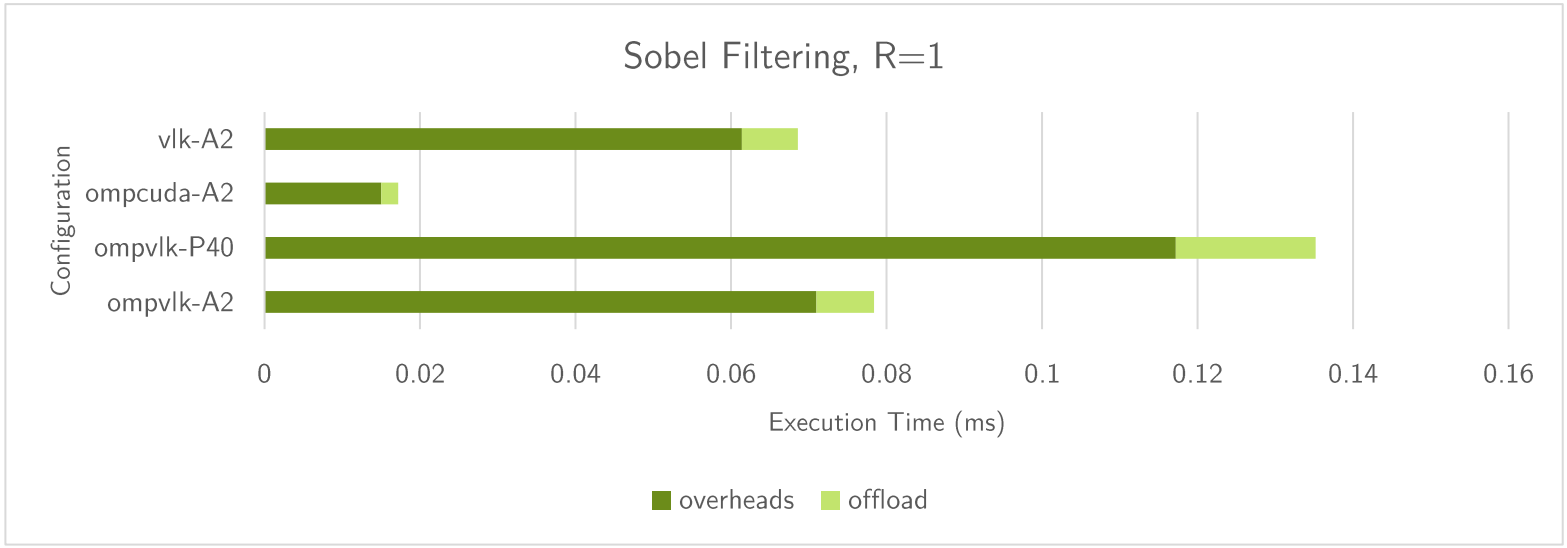


Image kernels – Results (cont.)





Lines of code comparison

We also compared our infrastructure to hand-crafted Vulkan versions of the benchmarks, in terms of the LOC metric

	adam	bfs	kuwahara	gaussian	sharpening	sobel
OpenMP	20	40	40	50	60	30
Shader(s)	100	30	70	60	90	80
Vulkan API code	270	240	240	250	280	280
Reduction	**94.6%**	**85.2%**	**87.1%**	**83.9%**	**83.8%**	**91.7%**

Conclusion & future work

Conclusion

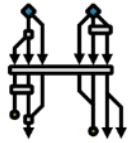
- We designed and implemented a novel, open source compiler and runtime infrastructure to support offloading to Vulkan-enabled GPUs through OpenMP
  - A translator converts code associated with target constructs to shader code and necessary host CPU code
  - A runtime Vulkan offloading plugin encapsulates the complete Vulkan Compute pipeline setup
    - Also provides OpenMP support within the offloaded code
- Our work is based on the open source OMPi compiler
  - https://paragroup.cse.uoi.gr/wpsite/software/ompi
  - Included in the upcoming release (stay tuned!)

Conclusion (cont.)

- Our results show that our infrastructure can exploit the parallelism potential of multiple GPU architectures, while hiding low-level API complexity
  - In general, it follows the OpenMP-to-CUDA offloading (ompcuda) and even outperforms it for larger problem sizes
    - The overhead is the main contributor, however it stays constant
    - Offload times are comparable
  - It can also match the performance of pure Vulkan code
    - While achieving ease-of-programming

Future work

- Future work includes:
  - Additional features (e.g., new loop scheduling strategies)
  - Optimization of the generated shader code in terms of size and performance
  - Tuning of Vulkan device plugin to minimize overheads
  - Improved buffer setup (according to shader variables)



HLPP 2025
18th International Symposium on
High-level Parallel Programming and Applications

Thank you for your time!

https://paragroup.cse.uoi.gr/wpsite/software/ompi

High-level Programming of Vulkan-based GPUs through OpenMP

Ilias K. Kasmeridis and Vassilios V. Dimakopoulos