Kernel Fusion in OpenCL

John A. Stratton[1,2(✉)], Jyothi Krishna V. S.[2], Jeevitha Palanisamy[2], and Karthikadevi Chinnaraju[2]

[1] Whitman College, Walla Walla, WA 99362, USA
strattja@whitman.edu
[2] MulticoreWare Inc., Chennai, Tamil Nadu, India
{jkrishna,jeevitha,karthikadevi}@multicorewareinc.com

Abstract. Kernel Fusion is a widely applicable optimization for numerical libraries on heterogeneous systems. However, most automated systems capable of performing the optimization require changes to software development practices, through language extensions or constraints on software organization and compilation. This makes such techniques inapplicable for preexisting software in a language like OpenCL.

This work introduces an implementation of kernel fusion that can be deployed fully within the defined role of the OpenCL library implementation. This means that programmers with no explicit intervention, or even precompiled OpenCL applications, could utilize the optimization. Despite the lack of explicit programmer effort, our compiler was able to deliver an average of 12.3% speedup over a range of applicable benchmarks on a target CPU platform.

Keywords: OpenCL · Kernel fusion · Heterogeneous computing

1 Introduction

Good software design practices and good software performance practices are sometimes in tension. One tension exists between the desire for software portability and the optimization opportunities available when the target architecture is known ahead of time to the developer. OpenCL was designed to address this tension by deferring code generation of accelerator machine code until application execution, allowing each accelerator vendor to build appropriate optimizations into the compiler for that particular accelerator. Another common tension is between the design principle of modularity and the optimization opportunities enabled by interprocedural optimization. Function encapsulation allows useful units of code to be reused in a variety of different contexts. However, in any particular use case, performance would likely benefit from optimizations taking into account the particular mixture of functions used in that circumstance.

Recent efforts have turned towards generative frameworks where the host application explicitly constructs, at run-time, a complete description of the computation to be performed on the accelerator [2,2,13,14]. Such strategies are effective, but require applications to be rewritten for the newly developed language.

In this work, we show how it is possible to implement inter-kernel optimizations entirely behind the OpenCL API. This strategy enables legacy software written with OpenCL C or already compiled into SPIR IR to take advantage of inter-kernel optimizations such as kernel fusion. While the extent of those optimizations can be limited by the information hidden by the narrow API, we find that in many examples, there is sufficient information for substantial performance improvements.

In this paper, we describe how the asynchronous nature of the OpenCL API can be exploited to perform inter-kernel optimizations without explicit intervention from the application programmer. Sections 2 and 3 cover the technical details of the opportunity and exploitation, respectively, of inter-kernel information in an OpenCL compiler through the OpenCL API. Section 4 show performance results demonstrating the impact of kernel fusion implemented with this strategy on a variety of applications. We conclude with a summary of related work in Sect. 5 and some final remarks in Sect. 6.

2 Background

While OpenCL is a widely supported standard, few implementations are available in open source. Portable OpenCL (POCL [8]) is one such actively supported open-source framework, so we will present our work in the context of that infrastructure. The POCL system is capable of executing OpenCL workloads on CPUs and supported GPUs with the main branch, and has been customized to a variety of other targets. This means that the architecture of POCL is likely to be representative of other OpenCL implementations as well, as many features are required by the OpenCL specification itself, as well as the constraints of discrete accelerators.

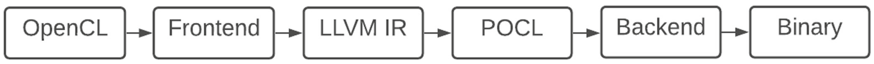

Fig. 1. POCL compilation chain.

The POCL implementation is divided into two layers, the host and device layer. The Host layer implements the portion of the OpenCL runtime that runs synchronously with the host application code. For our purposes, we will focus on the kernel compilation and execution portions of the system. The Host layer of the compiler comprises generic LLVM passes and optimization and is agnostic of the target hardware. The device layers include target specific implementations such as LLVM codegen and resource management that ensures proper sharing and synchronization of memory. The Fig. 1 shows the compilation path with POCL framework.

The POCL compiler uses clang as its frontend to generate LLVM IR or Standard Portable Intermediate Representation (SPIR [10]). POCL then proceeds by linking and inlining a target-specific builtin function library to the kernel. While targeting GPUs, POCL directly lowers the input OpenCL source code into SPIR code to target assembly code, as the GPU hardware and driver will manage the parallel work-item execution. The regions between barriers, will be executed by all work items before proceeding to the next region, are generated by the parallel region formation passes in POCL. For CPUs, POCL performs thread coarsening, instantiating explicit loops over work-item indexes within a work-group for those regions. POCL then adds target-specific optimizations such as vectorization and unrolling parallel regions. In the end, what was an LLVM-IR representation of the work of a single work-item becomes an LLVM-IR function encapsulating the computation of an entire work-group.

After compilation, when the application enqueues a kernel launch, the POCL compiler puts this command into its command queue to be sent to the asynchronously operating Device-runtime layer. For a CPU target, the device layer of the runtime manages the worker thread pool for parallel execution. The runtime dispatches work units to that worker pool when a kernel launch command is read from the queue. Even though some devices, such as the parallel cores of the host CPU itself, are capable of executing synchronously, the OpenCL standard mandates asynchrony between the host and device sides of the command queue. This design is beneficial for many systems, and can be exploited by our system to take advantage of the fact that kernels do not need to be eagerly executed when queued.

3 Kernel Fusion

In kernel fusion [2,3], we merge the code from multiple kernels to execute the code together as a single kernel. This process is expected to improve memory locality and can enable further instruction optimizations that only become apparent when optimizing the code across multiple kernels. In this section we describe our kernel fusion method. One of the salient features of our kernel-fusion framework is that our can perform kernel fusion even in the presence of loop-carried dependencies across multiple kernels if the compiler can determine the dependence to be bounded and deterministic.

Our entire fusion process is divided into two branches:

- **BRANCH A:** Which deals with total fusion. Here the participating kernels are either independent (i.e. they do not share I/O buffers) or there is no inter-work-item dependency. In such cases, the work of a given work-item in the first kernel can be immediately followed by the work of the work-item with the same index in the subsequent kernel, with no change in the computed results. The framework merges the kernels directly without any changes in kernel code or scheduling. The fusion framework makes use of the default POCL asynchronous scheduling.
- **BRANCH B:** deals with kernels with a non-zero loop-carry dependence. POCL kernels imply a global memory barrier in between them. This normally ensures all dependencies across the kernels are met. Our fusion framework

must ensure that the dependent work-items (or iterations) in the latter kernel are executed after all work-items from the former kernel on which they depend have been executed. The framework transforms the kernel code of the former kernel to merge the latest dependent iteration (for i-th iteration of the second kernel) of the kernel the i-th iteration of the latter kernel. The fused kernel, in such situation, mimics software pipelining in traditional loop fusion. In such cases, a few iterations of the predecessor kernel is executed before the merged kernel is executed, similar to how a software-pipelined loop may need to execute some startup iterations. Likewise, some of the work-items of the successor kernels will need to be executed after the merged kernel is executed. The number of iterations of the individual kernels and merged kernel executed will depend on scheduling and work-group sizes. In such cases, we transform the scheduling code to use work groups as large as possible to reduce the number of iterations of individual kernels executed. The framework reject the kernels as fusion candidates if a transformation to meet the dependencies cannot be applied.

Algorithm 1. Kernel Fusion algorithm. The *merged_kernel* list will contain the merged kernel along with the dependent iterations of individual kernels. We modify the *LAUNCH* function of the baseline POCL to check for inter-iteration dependency between the cached kernels.

```
 1: procedure LAUNCH(stack)    ▷ Launches with mergeable kernels in the stack
 2:     $merged\_kernel \leftarrow empty$
 3:     $carried\_dependency \leftarrow none$    ▷ Holds the dependency of merged kernel
 4:     $arg\_list \leftarrow empty$    ▷ Holds the argument list of merged kernel
 5:     for each kernel in stack do
 6:         $kernel\_args \leftarrow get\_kernel\_arguments(kernel)$
 7:         $kernel\_dependency \leftarrow calculate\_dependency(kernel)$
 8:         if $carried\_dependency = none$ then
 9:             $updated\_kernel \leftarrow kernel$
10:         else
11:             $updated\_kernel \leftarrow update\_kernel(kernel, carried\_dependency)$
12:         end if
13:         $merge\_kernels(merged\_kernel, updated\_kernel, argList)$
14:         $update\_carried\_dependency(carried\_dependency, kernel\_dependency)$
15:         $update\_argList(argList, kernel\_args)$
16:     end for
17: end procedure
```

Figure 2 describes our fusion framework. Our fusion framework extends from the compilation which handles the code transformation and merged kernel creation. The scheduling section requires some additional support to ensure the dependent iterations of individual kernels are executed for loop-carry dependent fusion. This scheduling is not used for total fusion to reduce the overhead in scheduling.

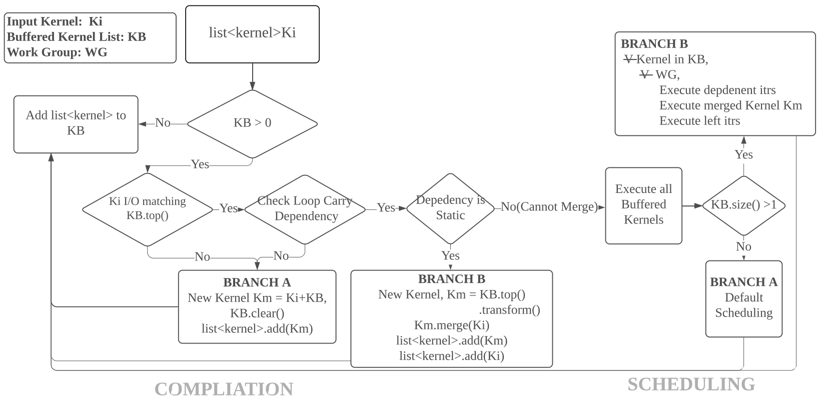

Fig. 2. The workflow of our fusion framework. The framework has two sections. The compilation section which handles the code transformation and merged kernel creation and the scheduling section which realizes the iteration scheduling for individual kernels and the merged kernel to handle the loop carry dependencies.

In our fusion compilation section the compiler follows a lazy loading policy. The compiler maintains a kernel buffer (KB) which will contain the kernels whose execution has been deferred so that fusion opportunities can be identified. When the first kernel received, it is directly moved from the command queue to the KB. For each additional kernel received, it is compared to the kernels already in the KB to determine whether the new kernel could be fused with the deferred kernels. If so, the fusion is performed, and the results of the fusion are put back in the KB in place of the kernels that were fused, potentially capable of being fused again with subsequent kernels. If not, or if a synchronization event is detected, all deferred and fused kernels are immediately executed with the appropriate scheduler.

To determine whether fusion is possible, for each new kernel, the compiler checks if there is a loop carry dependency with the previous kernels in the KB. The compiler also checks for the total number of iterations/work-items. If the total number of iterations are different, it considers the two kernels as non mergeable. Otherwise, if the kernels are fully independent (i.e. I/O buffers of the previous kernels is different from the I/O of the new kernels) or if there is no loop-carry dependency, the compiler will perform total fusion. In such cases, the compiler merges the incoming kernel with the existing kernel in the KB. The new merged kernel replaces the existing kernel in the KB. As the framework is able to achieve total fusion, the framework only schedules the merged kernel in the execution buffer. The merged kernel will be using the default POCL scheduling and the work-groups.

Branch B shows the path taken while merging kernels with a constant loop carry dependency. In such cases, the framework will not be able to do a complete fusion of the kernels. The compiler will find an upper bound for the loop-carry dependency distance and make the necessary transformation to the predecessor kernels before merging. The last dependent iteration of the predecessor kernel is merged with the iteration of the incoming kernel. The compiler will maintain all the individual kernels along with the merged kernel in the execution buffer. The compiler will also maintain the relative loop-carry dependence list for each kernel in the execution queue. The run-time module of Branch B will execute the dependent iterations of the independent kernels before it starts executing the merged kernel in the work group. The actual dependent-iterations will be contingent on the number of work-groups created. As more work-groups are created, the more fragmented the scheduling becomes and more individual dependent iterations needs to be executed. The default POCL scheduling for pthreads is dynamic with scheduling chunk size determined by the function $\min(32, N/num\_threads)$ where N stands for max iterations and $num\_threads$ gives the total number of parallel threads. This limits the number for iterations executed in a work-group to 32. We need the size of the work-group to be as large as possible so that we execute the fewest individual iterations. Consequently, we use

$$(N - max\_loop\_dependence)/num\_threads \tag{1}$$

as our work-group size. Here $max\_loop\_dependence$ stands for the maximum loop carry dependence across all merged kernels.

3.1 Loop Carry Dependent Fusion

Figure 3 explains the three stages involved in loop carry dependency fusion. The example contains three kernels (K_1, K_2 and K_3). Every iteration of K_2 has a loop carry dependency of a single iteration over K_1, and K_3 has a loop carry dependency of one iteration over K_2 The first stage in compilation involves identification of these dependencies using LLVM [6,11] analysis passes. Once the dependency is determined to be static, in the second stage the compiler transforms Kernel $K1$ to satisfy the dependency with $K2$ by simply updating the iterators used in the kernel. The compiler then merges the transformed kernel K_1‘ with $K2$ to create a temporary merged kernel K_{12}. In the next iteration K_{12} is transformed to satisfy the dependency with K_3 before creating the final merged kernel K_{123}. We maintain individual Kernels ($K1, K2, K3$) and the merged kernel K_{123} in the execution queue.

In scheduling stage, the framework divide the iterations equally among all threads. Each thread will have its own iteration queue which will contain all kernels. Each will execute the dependent iterations of Kernels K_1 and K_2 for the iterations allocated to the thread. Once the dependents iterations are executed the allocated merged kernel iterations are scheduled.

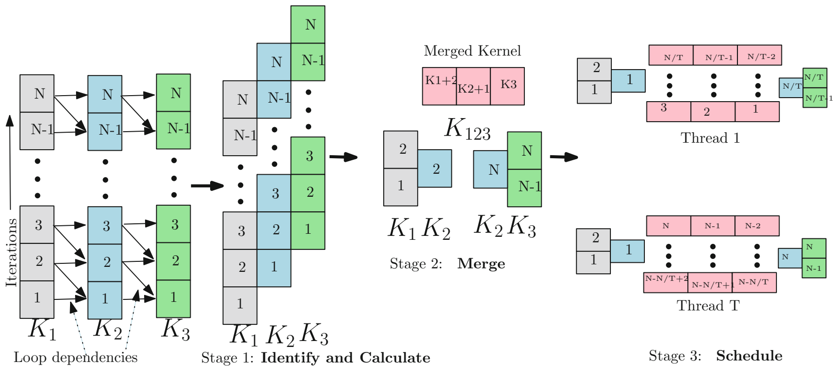

Fig. 3. Loop carry dependency Fusion. In Stage 1, we identify these dependency and if the value is deterministic we mark the kernels for fusion. In Stage 2, we try to merge the larges non-dependent chunks of Kernels K_1, K_2 and K_3 to create a new merged kernel K_{123}. In stage 3, scheduling For each thread i We will run all dependent iterations of Kernel K_1 and K_2 before executing the chunk of merged Kernel K_{123}.

3.2 Other Implementation Details

We have implemented our framework on top of POCL version 1.5. POCL has defined a set of optimizations such as barrier removal and autovectorization [7,8] for OpenCL kernels. Our Merge module is inserted before the these passes.

The overhead of analysis and code generation would substantially increase the apparent execution time of a set of kernels, given the relative complexity of the operations. However, many applications will regularly execute the same sequence of kernels with the same data dependencies, rather than shut down and start up the OpenCL environment for every new buffer. This is particularly applicable for applications processing consecutive frames of a video or a batch of images in a machine learning inference query. To take advantage of this, the first time a kernel is fused in the system, the fused kernel is cached and re-referenced when the same kernels with the same dependencies are seen again. This significantly decreases the effective cost of our transformations, but does not complete eliminate the runtime checking needed to verify the buffer dependencies between kernels in the queue every time.

We find that the default scheduling algorithm of POCL introduces a lot of fragmentation, which is counterproductive to the kernel merge. As a result, we were observing a lot of overhead in scheduling the kernels multiple times. We overcome this issue by introducing a new low-overhead scheduling based on the number of parallel threads (Function 1). This will also keep the number of iterations of individual kernels executed to minimum and we can extract more performance from the merged kernels. However, in such cases we are trading off the advantages of dynamic scheduling.

4 Results

In this section we discuss the efficacy of our fusion framework on POCL version 1.5. We demonstrate the efficacy of our fusion framework in the CPU backend of POCL. However, our framework can be easily ported to other OpenCL supported devices as only the scheduling module of the framework is device-specific.

Table 1. List of benchmarks used for evaluation, taken from a sampling of image processing operations.

Name (key)	Description	No of kernels	
		Before fusion	After fusion
Image Negatives (`ImN`)	Takes the negative of input image and performs jitter operation on it	3	2
Image Enhancement (`ImE`)	Enhance the input image using brightening and non max suppression filter	2	1
Intensity Transformation (`Int`)	Reduce the brightness of image, corrects it and modifies the color	3	1
Log Transformation `Log`	Log transform followed by grey-scaling the image and finally binarization of the image	4	1
Spatial Filtering (`SpF`)	It is a edge detection algorithm that uses convolutional filter	4	2
RGB2YUV (`R2Y`)	Converts an RGB image in interleaved format to a YUV image in interleaved format	3	1
HueContrastCrop (`Con`)	Modifies the input image color and crops it	3	1
Morphological Transforms (MT)	Dilate and erode a part of image	3	1
Gaussian (Gau)	Applying Gaussian Filter to the image	3	1
Canny Edge Detector (Can)	Applying Canny Edge detector algorithm on image	8	1
Image Transformation (`ImT`)	Apply log transformation on input image and convert them to single-channel grayscale. Performs binarization and crop the input image	4	2

We have tested our framework with a set of image processing benchmarks. Table 1 describes the set of benchmarks we have used for testing. In `R2Y` benchmark we do not have inter-iteration dependency across kernels. So we are able to do complete fusion for `R2Y`.

Our test machine is build with 16 GB RAM and AMD Ryzen 5 3600x with 6 cores and 12 threads. We used LLVM version 9 with Ubuntu 18.04. Table 2 provides the execution time of baseline POCL and POCL with kernel fusion in seconds. We observe an average improvement of 12.30% in overall execution time over the set of benchmarks.

Table 2. Execution time of various benchmarks in baseline POCL and POCL with kernel fusion in seconds.

Benchmark	Baseline POCL in s	Fused kernels in s
ImN	3.85	3.04
ImE	0.10	0.12
Int	17.59	16.64
Log	3.68	3.28
SpF	1.38	1.25
R2Y	2.94	1.65
Con	3.38	3.35
MT	4.08	4.20
Gau	1.50	1.11
Can	6.93	5.21

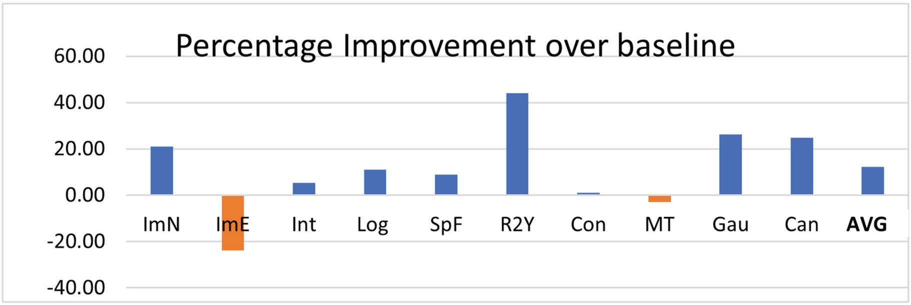

Fig. 4. Percentage improvement of our fusion compiler when compared to POCL baseline.

Figure 4 shows the percentage improvement in execution time for fusion framework over the baseline POCL. Some benchmarks show significant improvements, such as the `R2Y` benchmark where complete kernel fusion is possible, resulting in a 44% improvement in execution time. We can also see the overhead of our framework show in benchmarks with little potential from fusion and short kernels, such as the `ImE` benchmark. As a result of the additional overhead, we see a 20% increase in execution time for `ImE`. We can also the that kernel fusion is typically a memory locality optimization, so arithmetic-intensive benchmarks such as `Int` see little performance improvement. Other benchmarks generally saw improvements, with the magnitude of those improvements varying with the amount of kernel fusion opportunity and potential benefit, and the fixed overhead relative to the execution time of each individual kernel.

5 Related Works

Kernel Fusion is a widely known and extensively employed technique to improve execution time and reduce power consumption [1–3,12,15,17]. In [15] authors explain a kernel fusion framework for independent kernels in GPU to improve the power consumption. [2] explains a framework to automatically fuse GPU OpenCL kernels for Stan Math Library, using language extensions to drive the fusion optimizations. The framework was able to achieve performance comparable to the hand tuned fusion kernels. In [17] authors explain a runtime framework which will decide which kernel merges would result in faster code. The learning framework is based on the number of branches executed by each kernel which is especially costly in streaming processors such as GPUs. The OpenCL fusion described in [3] explains a framework to perform kernel fusion in CPUs where they explain data fusion for flow dependent kernels. In this work we have mostly concentrated on the data fusion with inter loop dependencies, however our framework can handle non dependent kernels. There has been a few compiler driven works for GPU devices [1,3–5], for SYCL cross-platform framework [12] for CUDA supported devices [1,4] to improve the performance across the kernels

In our work we are introducing a framework to transform loop carry dependent kernels to perform aggressive fusion in CPU. There have been some works which rely on scheduling [9,16] and avoid static kernel fusion to improve the performance and/or energy consumption. In [9], the authors describe three different concurrent execution of multiple kernels. The inner thread execution method is similar to the total fusion we elaborate as every thread in the warp executes the corresponding iteration of kernels. In [16] the authors introduces a framework to preempt a kernel at thread-block level to allow simultaneous execution of multiple kernels.

6 Conclusion

In this paper we described our kernel fusion framework in POCL. If the input kernels has no inter-loop dependencies, our framework performs a total fusion. When there is inter-loop dependencies our framework divides the input kernels to dependent iterations and independent iterations. The framework then transforms the independent iterations and fuses the kernels. The scheduling part of our framework executes the dependent iterations of the individual kernels before executing the fused kernel. We compared the performance of our kernel fusion framework and we observed an overall execution time improvement of 12.30% over the baseline POCL implementation. In future we want to optimize our fused kernel by eliminating the intermediate buffers which were primarily used to transfer data from one kernel to the next kernel in the baseline implementation. We would also want to implement compiler flags that a user could invoke to override the default behavior of the compiler, if the user can predict that kernel fusion is possible but detrimental. The will ensure that the users can weigh in to reduce the chances of the overhead dominating the gains of kernel fusion.

References

1. Aliaga, J.I., Pérez, J., Quintana-Ortí, E.S.: Systematic fusion of CUDA kernels for iterative sparse linear system solvers. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 675–686. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_52
2. Ciglarič, T., Češnovar, R., Štrumbelj, E.: Automated OpenCL GPU kernel fusion for Stan math. In: Proceedings of the International Workshop on OpenCL, IWOCL 2020. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3388333.3388654
3. Filipovic, J., Benkner, S.: OpenCL kernel fusion for GPU, Xeon Phi and CPU. In: 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 98–105 (2015). https://doi.org/10.1109/SBAC-PAD.2015.29
4. Filipovič, J., Madzin, M., Fousek, J., Matyska, L.: Optimizing CUDA code by kernel fusion: application on BLAS. J. Supercomput. **71**(10), 3934–3957 (2015). https://doi.org/10.1007/s11227-015-1483-z
5. Gong, X., Chen, Z., Ziabari, A.K., Ubal, R., Kaeli, D.: TwinKernels: an execution model to improve GPU hardware scheduling at compile time. In: 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 39–49 (2017). https://doi.org/10.1109/CGO.2017.7863727
6. Jääskeläinen, P.O., de La Lama, C.S., Huerta, P., Takala, J.H.: OpenCL-based design methodology for application-specific processors. In: 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 223–230 (2010). https://doi.org/10.1109/ICSAMOS.2010.5642061
7. Jääskeläinen, P., et al.: Exploiting task parallelism with OpenCL: a case study. J. Signal Process. Syst. **91**, 1–14 (2019)
8. Jääskeläinen, P., de La Lama, C.S., Schnetter, E., Raiskila, K., Takala, J., Berg, H.: POCL: a performance-portable OpenCL implementation. Int. J. Parallel Prog. **43**(5), 752–785 (2014). https://doi.org/10.1007/s10766-014-0320-y
9. Jiao, Q., Lu, M., Huynh, H.P., Mitra, T.: Improving GPGPU energy-efficiency through concurrent kernel execution and DVFs. In: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1–11 (2015). https://doi.org/10.1109/CGO.2015.7054182
10. Kessenich, J., Ouriel, B., Krisch, R.: SPIR-V specification (2021)
11. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California, March 2004
12. Potter, R., Keir, P., Bradford, R.J., Murray, A.: Kernel composition in SYCL. In: Proceedings of the 3rd International Workshop on OpenCL, IWOCL 2015. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2791321.2791332
13. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York (2013). https://doi.org/10.1145/2491956.2462176
14. Rotem, N., et al.: Glow: graph lowering compiler techniques for neural networks. arXiv preprint arXiv:1805.00907 (2018)

15. Wang, G., Lin, Y., Yi, W.: Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In: 2010 IEEE/ACM International Conference on Green Computing and Communications International Conference on Cyber, Physical and Social Computing, pp. 344–350 (2010). https://doi.org/10.1109/GreenCom-CPSCom.2010.102
16. Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., Guo, M.: Simultaneous multikernel GPU: multi-tasking throughput processors via fine-grained sharing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 358–369 (2016). https://doi.org/10.1109/HPCA.2016.7446078
17. Wen, Y., O'Boyle, M.F.: Merge or separate? Multi-job scheduling for OpenCL kernels on CPU/GPU platforms. In: Proceedings of the General Purpose GPUs, GPGPU-10, pp. 22–31. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3038228.3038235