Multi-Variant User Functions for Platform-Aware Skeleton Programming

August ERNSTSSON [a] and Christoph KESSLER [a 1]
[a] *PELAB, Dept. of Computer and Information Science*
Linköping University, 581 83 Linköping, Sweden

Abstract. Today's computer architectures are increasingly specialized and heterogeneous configurations of computational units are common. To provide efficient programming of these systems while still achieving good performance, including performance portability across platforms, high-level parallel programming libraries and tool-chains are used, such as the skeleton programming framework SkePU. SkePU works on heterogeneous systems by automatically generating program components, "user functions", for multiple different execution units in the system, such as CPU and GPU, from a high-level C++ program. This work extends this multi-backend approach by providing the possibility for the programmer to provide additional variants of these user functions tailored for different scenarios, such as platform constraints. This paper introduces the overall approach of multi-variant user functions, provides several use cases including explicit SIMD vectorization for supported hardware, and evaluates the result of these optimizations that can be achieved using this extension.

Keywords. Skeleton programming, SkePU, Heterogeneous computing, Multi-variant user functions, Vectorization

1. Introduction

Programming of complex multi-core and heterogeneous computer architectures can be a difficult task, especially when there is a desire to fully and efficiently utilize the available processing resources. Managing the required workload distribution, synchronisation, and data management often requires expert knowledge and long-time experience. This is especially true if also *performance portability* is desired, as different systems can vary widely in terms of both the number and types of processing cores, as well as in other characteristics such as memory hierarchy.

High-level parallel programming frameworks aim to improve on this situation by reducing the user-facing complexity of programs. A small number of highly optimized but still general programming building blocks are presented through a high-level interface. This category of frameworks include application specific languages, PGAS (Partitioned Global Address Space) interfaces, dataflow models, and more, but most importantly for this paper: the *skeleton programming* [4] concept, borrowing the higher-order operations of functional programming such as `map` and `reduce`, and implemented as an abstraction level that is portable across both multi-core and heterogeneous computers and larger supercomputer clusters. Skeleton programming uses generic building blocks encoding

common computational *patterns* as the high-level programming interface. Examples of such patterns are often divided into two categories: *data parallel* patterns such as the aforementioned map and reduce, and *task parallel* patterns including task farming and parallel divide-and-conquer, among others.

The core contribution of this paper is a generalization of the variant selection mechanism for the skeleton programming framework SkePU, where the problem-specific, sequential user code used to customize a skeleton at skeleton instantiation can be provided in several variants, some of which might even be platform-specific. This is done in a general-purpose programming environment, which differentiates the approach from existing domain-specific variant selection [8]. Our work is also tightly integrated with a platform modeling system [10] allowing build-time lookup of eligible variants going beyond only algorithmic choice or minor variations in performance tuning parameters. The approach is powerful and flexible enough to allow selection based on hardware architecture, levels of heterogeneity, software installations, and more.

Relevant background on SkePU is introduced in Section 2, followed by the idea and implementation of the core contribution in Section 3. We present several use cases in Section 4 and experimental evaluation results in Section 5. We discuss related work in Section 6 and conclusions and future work in Section 7.

2. Background: SkePU

One example of a skeleton programming framework is *SkePU*[2] [6, 7], a C++ compiler toolchain and runtime library implementing data-parallel skeleton programming. SkePU targets heterogeneous systems with multiple *backends*, such as multi-core CPU, GPUs using either CUDA or OpenCL, or even a "hybrid" combination of several backends at once [15]. Each skeleton pattern defined in SkePU (one of `Map`, `Reduce`, `MapReduce`, `Scan`, or `MapOverlap`) is instantiated with a *user function*, typically a small piece of code that is applied once for each element in the input data at run-time. This creates a skeleton instance that can be called like a normal C++ function, but internally provides automatic backend selection and data management (using *smart containers*) across backends.

The SkePU framework performs source code analysis of the input program by an external source-to-source compiler. This tool locates SkePU skeleton instances used by the programmer, and identifies the user functions that are needed to instantiate them. The parameter signature and body of these functions is used to generate fully instantiated backend wrappers and kernels for each user function and skeleton instance (including distinct source files for GPU execution). Each user function therefore has a number of (implicit) *implementation variants* available for it, and the runtime library part of SkePU will select automatically among these variants at program execution time.

The main contribution of this paper is the extension of the variant selection system in SkePU to be a *user-facing* feature of the framework, as presented in the next section.

[2] http://www.ida.liu.se/labs/pelab/skepu/

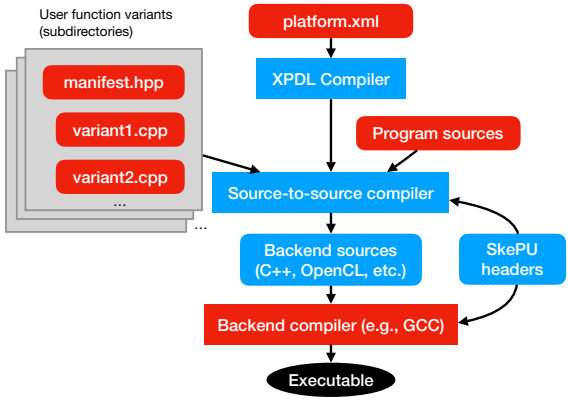

Figure 1. Overview of the components involved in SkePU variant selection and subsequent build process.

3. Idea and Implementation

There are multiple scenarios where a user function with a singular definition can be too restrictive for the purposes of performance: use cases include algorithms with different tradeoffs in time complexity versus memory complexity (some platforms may have very limited memory space available per execution thread), instruction set architecture differences such as native double or half precision floating point arithmetics, the existence of SIMD vector instructions, or other hardware-accelerated implementations of common computations. Since these attributes are *constrained* on the underlying platform, the software-defined code variants must somehow be declared compatible only with the appropriate hardware configurations. For this we employ a combination of *language attributes*, annotations at source-code level that are recognized by the SkePU source-to-source compiler, in addition to the *platform description language* XPDL [10].

A platform description (such as the one given in Listing 1) is supplied to the SkePU source-to-source compiler and depending on the attributes in the model, user function variants are either included or removed from the resulting program. In this example, the user function variant in Listing 3 requires the *Intel AVX* extension to the instruction set. The list of variants for each user function and their prerequisites for inclusion are declared in a *manifest file* (example given in Listing 4). Here XPDL metaprogramming queries or other statically evaluated expressions can be used. As the model in Listing 1 declares the platform to support this extension (line 7 in Listing 1), this vectorized variant will be included for variant selection at run-time. In cases where library or binary compatibility is not required for the extension, this filtering of eligible variants can also happen at run-time, as long as the XPDL model is available for querying. This approach is preferred when a single program executable might run on different hardware configurations.

User function variants are defined externally from the main source file. The variants are placed in individual source files in subdirectories, following a standard naming schema, with one directory for each user function. A *component implementation descriptor* file defines the hardware platform and run-time requirements for each variant. See Figure 1 for an illustration of the workflow: the outlined rectangles denote directories in the file system and the filled rectangles represent files.

Listing 1: XPDL model for an Intel Xeon Gold 6130 CPU. Please refer to XPDL publications [10] and documentation for details about the syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<xpdl:model xmlns:xpdl="http://www.xpdl.com/xpdl_cpu"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.xpdl.com/xpdl_cpu xpdl_cpu.xsd ">
  <xpdl:component type="cpu" />
  <xpdl:cpu name="Intel_Xeon_Gold_6130" num_of_cores="16"
    num_of_threads="32" isa_extensions="avx avx2">
  <xpdl:group prefix="core_group" quantity="16">
    <xpdl:core frequency="2.1" unit="GHz" />
    <xpdl:cache name="L1" size="32" unit="KiB" set="16" />
    <xpdl:cache name="L2" size="1" unit="MiB" set="16" />
  </xpdl:group>
  <xpdl:cache name="L3" size="22" unit="MiB" set="1" />
  <xpdl:power_model type="power_model_Gold_6130"></xpdl:power_model>
  </xpdl:cpu>
</xpdl:model>
```

4. Use Cases

In this section we present two use cases in detail: user function vectorization and multi-variant components with the `Call` skeleton. We also provide further examples for application of multi-variant components at the end of the section.

4.1. Vectorization Example

As an example of where user function variants are applicable, consider instruction set extensions for SIMD vectorization. These extensions allow the processor to compute the same instruction in parallel over multiple data items, even from a single thread. Many compilers today are *auto-vectorizing* [11–13], but this optimization requires a number of preconditions to be satisfied, such as the correct data alignment and no pointer aliasing; and even then, additional compiler flags are often required. For a high-level parallel program such as a SkePU application, aggressive inlining and loop unrolling must also be applied by the backend (external to SkePU) compiler before there is even an opportunity for auto-vectorization.

For the aforementioned reasons, vectorization is a good motivational use case for multi-variant user functions. Consider the SkePU program in Listing 2. The program performs element-wise addition of two vectors using the SkePU `Map` skeleton with arity 2. The user function `add` is trivial, with two inputs (one from each vector) and the function body returning the sum of the two elements. This user function is straight-forward for the SkePU source-to-source compiler to handle when generating output for all backends: sequential CPU, OpenMP, CUDA, and OpenCL; it is just a matter of copying the function body. However, by this approach, the CPU backends will not be guaranteed optimal performance in the case of the hardware platform supporting SIMD ISA extensions. As such, it makes sense to provide a variant of `add` and make it available for run-time selection.

Listing 2: A SkePU program performing element-wise vector addition.

```
float add(float a, float b) { return a + b; }

int main(int argc, char *argv[])
{
  const size_t size = N; // multiple of 8
  auto vector_sum = skepu2::Map<2>(add);
  skepu2::Vector<float> v1(size), v2(size), res(size);
  vector_sum(res, v1, v2);
}
```

Listing 3: Variant of the `add` user function with explicit vectorization.

```
#pragma skepu vectorize 8
void add(float* c, const float *a, const float *b)
{
  __m256 av = _mm256_load_ps(a);
  __m256 bv = _mm256_load_ps(b);
  __m256 cv = _mm256_add_ps(av, bv);
  _mm256_store_ps(c, cv); // return by pointer
}
```

Listing 4: Manifest file for user function `add`.

```
skepu::VariantList {
  skepu::Variant("add_avx",
    skepu::Requires(
      xpdl::includes<xpdl::cpu_1::isa_extensions, xpdl_avx>::value
    ), skepu::Backend::Type::CPU
  )
};
```

Listing 3 contains a variant of `add` that is defined in a separate file as outlined in Section 3. This file is referenced from the manifest, as seen in Listing 4. In this case, there needs to be a *block* of eight elements available for the function to enable the use of SIMD instructions, which is different in signature to the default variant.[3] This variant uses compiler intrinsic functions which map directly to Intel AVX instructions. The elements in this variant are passed and returned by pointer, and the component implementation descriptor contains the specification of how many elements it accepts in one block (here illustrated by an inline `pragma`). The elements in the array have to be copied to intermediate vector registers before computation.

[3]The need for framework support in this example is not a universal trait; user function variants can be defined with the same signature and even without any required platform constraints.

4.2. Generalized Multi-variant Components with the Call Skeleton

The version 2 revision of SkePU [7] introduced an atypical skeleton construct known as `Call`. The Call skeleton, unlike all other skeleton constructs in SkePU and other typical skeleton programming libraries, does not encode a computational pattern, but rather is an entry point for a self-contained *component* for arbitrary computations. This construct is highly useful in SkePU for two main reasons: firstly, not all computations can be efficiently expressed as data-parallel algorithms, which is the type of patterns present in SkePU, and it is desirable to let generic computations integrate with the smart container and backend selection and tuning systems within SkePU. Secondly, the optimal way to structure computations is in general different for different parallel backends; there needs to be a way to provide *variants* also for these non-skeleton computations.

A common class of computations that fit the above criteria are sorting algorithms. Another example is the fast Fourier transform (FFT) [19], which has several highly optimized implementations available at library level. In cases such as FFT, an instance of `Call` can instantiated with a naive sequential FFT algorithm as the default user function, and additional user function variants are specified as shown in Figure 1 and implemented as thin wrappers over libraries such as FFTW for CPU and CuFFT for Nvidia GPUs. Both the backend type and the presence of libraries in the target system is specified and taken into account for variant selection.

4.3. Other Use Cases

There are a number of other use cases for when multi-variant user functions can be useful for improving performance portability. Below are some suggestions: The user can specify a hand-optimized user function variant to be used only with a certain backend, such as CUDA (declared via the platform attribute in the user function's component implementation descriptor), while the generic auto-generated user function is used for all other backends. Even within the same backend and the same platform constraints, complex user functions may offer multiple variants implementing the same computation by different algorithmic approaches. Selection between the variants can be controlled by input size and shape, as well as other run-time properties such as idle resources and memory pressure. See e.g. the CellSort sorting algorithm [9] where the algorithm used is closely coupled to the characteristic architecture and instruction set of the Cell processor. When SkePU skeletons are invoked from a language other than C++, components that have a variant defined for that language would have lower overhead due to bridging and data representation and would open up for improved compiler optimization.

5. Performance Evaluation

We present performance evaluations for two distinct use cases for multi-variant user functions: vectorization of `Map`-type skeleton applications on real and complex numbers, and specialization of the algorithms used in the user function of a stencil-type image filtering operation using `MapOverlap`.

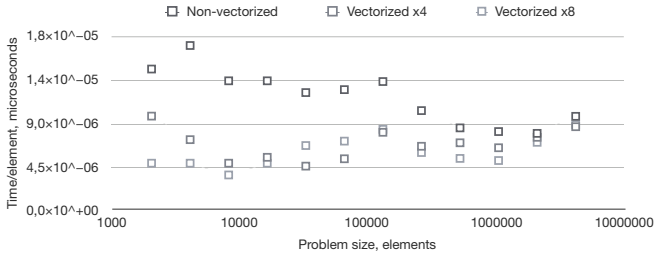

Figure 2. Element-wise vector addition, three variants. Execution time normalized (per element).

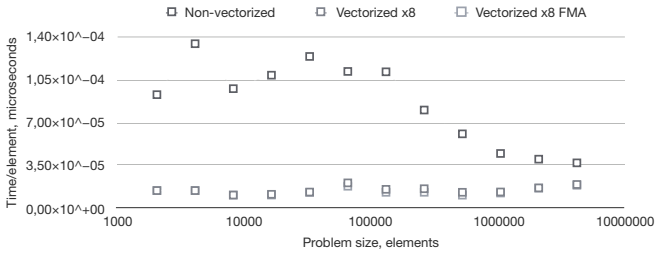

Figure 3. Element-wise complex vector multiplication, three variants. Execution time normalized (per element).

5.1. Vectorization

To demonstrate the performance gained from vectorization of user functions in a scenario in which automatic compiler optimization might be prohibited, we test the example from Section 4.1 using the Intel C++ Compiler v.18.0.1. `-O3` level optimization is enabled for all benchmarks, and the results are presented as the average of 100 runs. All computations are performed on single-precision floating point data. The target system uses Intel Xeon Gold 6130 processors. Two vectorization scenarios are evaluated:

Element-wise vector addition: Three variants are compared: no vectorization, and vectorization by a factor of four and eight, respectively.

Element-wise vector multiplication of complex numbers: Complex numbers stored in struct-of-arrays format, with four input data containers in total. Three versions are tested: no vectorization, factor eight direct vectorization, and a refactored vectorized version using fused multiply add (FMA) vector instructions.

For scalar element addition, the results show that there is always a benefit of vectorization if available. However, as seen in Figure 2 the overhead of loading and storing vector registers is significant when there is only one vector instruction to compute. The choice between four element vector instructions and eight element variants does not matter as much, as the best performer is inconsistent. It is clear that more computation is required to get the most out of manual vectorization.

We also evaluate complex number multiplication (Figure 3). The complex numbers are stored in cartesian form and multiplied element-wise according to $(a+bi)\times(c+di) = (ac-bd)+(ad+bc)i$. There are more vector instructions to amortize the register transfer overhead over in this case, even though the number of inputs is doubled. An alternate version with FMA instructions provides more efficient computation but at the cost of reducing this amortization factor.

Table 1. User function variants for median filtering.

Variant	Time complexity	Memory complexity	Dependencies
Double loop	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	None
Histogram	$\mathcal{O}(n + \|D\|)$	$\mathcal{O}(\|D\|)$	None
`qsort`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	C standard library

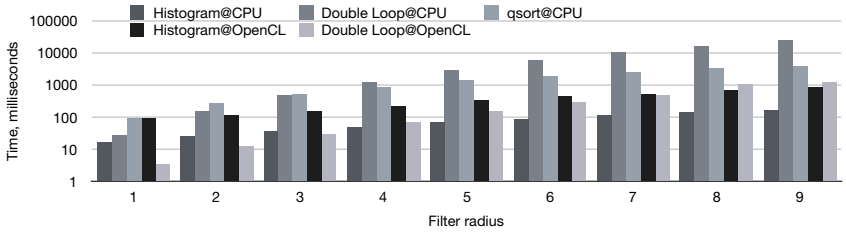

Figure 4. Median filtering using different median computation algorithms.

5.2. Median Filtering

To demonstrate and evaluate the application of multi-variant user functions to provide different algorithmic approaches to the same computation, we look at the median filtering operation on images. For each pixel in the output image, the filter selects the *median* value of all pixels in a region surrounding the corresponding pixel in the input image. The region is defined by a *radius*, the same in both x and y dimensions. Using the `MapOverlap` skeleton, the image filter is then implemented directly by providing the median-finding algorithm as the user function. This can be done in several ways: by sorting the elements in the region, brute-force counting search, or by a histogram collection, among others. The characteristics of the aforementioned three approaches are compared in Table 1 (in the table, n denotes input size and $|D|$ denotes the size of the value domain).

A comparison of execution times for the different variants is presented in Figure 4. The OpenCL variants target a single NVIDIA Tesla K20c GPU. The radius is varied in the range 1-9 pixels, but note that this has an effect in two dimensions and will scale the input region in the user function quadratically. The input image is fixed at 512×512 pixels, in 24-bit RGB format. The results show that there is no algorithm that is optimal across both backends; we even see that, on the GPU, the best variant varies with the filter radius.

6. Related Work

High-level parallel programming using skeletons or patterns [4] allows to model semantics as well as parallelization-relevant properties (such as type of parallelism, data access pattern, data locality constraints) of a computation using special predefined generic constructs (called skeletons or patterns) at a level of abstraction that is clearly above that of source code (such as OpenMP, OpenCL or CUDA). Existing skeleton programming frameworks include SkePU [6, 7], FastFlow [1], Marrow [14], GrPPI [5], Thrust [3] and others.

None of these skeleton programming frameworks considered automated, platform-specific operator specialization for multi-element groups in skeleton instantiations or calls. Lift, [18] on the other hand is a framework consisting of a functional pattern-based programming language, a compiler and an intermediate representation with pre-defined skeleton-like constructs for the hierarchical, functional modeling of data-parallel computations. It allows for (cost-model directed) rewriting of Lift IR trees by a design space exploration process to automatically take into account platform-specific structures such as SIMD operations, data transfers and data layout transformations, which can be expressed by OpenCL-specific constructs. While Lift is more general than our method, it requires the programmer to specify skeleton instances as a hierarchically nested functional decomposition of multiple primitive operators. In contrast, our approach is based on the simpler SkePU programming API, which is more high-level and does not require special tooling nor automated design space exploration nor an explicit intermediate representation.

PetaBricks is another framework which also exposes algorithmic variant ("choice") selection [2, 16]. In contrast to SkePU, PetaBricks is task-oriented with a more involved run-time scheduling system, and does not integrate a platform modeling subsystem into the toolflow.

It is also possible to take a more domain-specific approach. *SLinGen* [17] is a generative programming environment for linear algebra which outputs optimized C code, including optional vectorization driven by intrinsics. The *Cl1ck* system for matrix computations [8] focuses on generating multiple alternative application variants for a single operation.

The limitations of compiler auto-vectorization are explored by Larsen et al. [11] who also suggest improvements to the programming language and environment to facilitate the optimization in more scenarios.

7. Conclusions and future work

Introducing multi-variant user functions increases the performance portability aspect of SkePU programs by allowing the (expert) user to supply optimized source code for different target architectures. The extension is optional to use and not source breaking, and does not impact the programmability of the SkePU framework.

The multi-variant user functions is a part of the multi-variant component programming model developed within the EXA2PRO[4] project. Definition and declaration of user function variants will follow the general component declaration syntax in the EXA2PRO compilation workflow and, together with SkePU skeletons as components themselves, provide *nested* component selection in the EXA2PRO run-time system.

Acknowledgements

This work has been partly funded by the EU Horizon2020 project grant no. 801015 *EXA2PRO* and Swedish National Graduate School in Computer Science (CUGS).

[4] https://exa2pro.eu

References

[1] Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana, p. 13 (2012)

[2] Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: A language and compiler for algorithmic choice. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, pp. 38–49. ACM, New York, NY, USA (2009). DOI 10.1145/1542476.1542481

[3] Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for CUDA. GPU Computing Gems, Jade Edition (2011)

[4] Cole, M.I.: Algorithmic skeletons: Structured management of parallel computation. Pitman and MIT Press, Cambridge, Mass. (1989)

[5] del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurrency and Computation: Practice and Experience **29**(24), e4175 (2017). DOI 10.1002/cpe.4175. E4175 cpe.4175

[6] Enmyren, J., Kessler, C.W.: SkePU: A multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the fourth international workshop on High-level parallel programming and applications, pp. 5–14. ACM (2010)

[7] Ernstsson, A., Li, L., Kessler, C.: SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. International Journal of Parallel Programming pp. 1–19 (2017). DOI 10.1007/s10766-017-0490-5

[8] Fabregat-Traver, D., Bientinesi, P.: Automatic generation of loop-invariants for matrix operations. In: 2011 International Conference on Computational Science and Its Applications, pp. 82–92 (2011). DOI 10.1109/ICCSA.2011.41

[9] Gedik, B., Bordawekar, R.R., Yu, P.S.: Cellsort: High performance sorting on the cell processor. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, pp. 1286–1297. VLDB Endowment (2007)

[10] Kessler, C., Li, L., Atalar, A., Dobre, A.: XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization. In: Proc. 44th International Conference on Parallel Processing Workshops, ICPP-EMS Embedded Multicore Systems, in conjunction with ICPP-2015 (2015). DOI 10.1109/ICPPW.2015.17

[11] Larsen, P., Ladelsky, R., Lidman, J., McKee, S.A., Karlsson, S., Zaks, A.: Parallelizing more loops with compiler guided refactoring. In: 2012 41st International Conference on Parallel Processing, pp. 410–419 (2012). DOI 10.1109/ICPP.2012.48

[12] Levine, D., Callahan, D., Dongarra, J.: A comparative study of automatic vectorizing compilers. Parallel Computing **17**(10), 1223 – 1244 (1991). DOI https://doi.org/10.1016/S0167-8191(05)80035-3

[13] Maleki, S., Gao, Y., Garzarán, M.J., Wong, T., Padua, D.A.: An evaluation of vectorizing compilers. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, pp. 372–382 (2011). DOI 10.1109/PACT.2011.68

[14] Marques, R., Paulino, H., Alexandre, F., Medeiros, P.D.: Algorithmic skeleton framework for the orchestration of GPU computations. In: Euro-Par 2013 Parallel Processing, vol. LNCS 8097, pp. 874–885. Springer (2013)

[15] Öhberg, T., Ernstsson, A., Kessler, C.: Hybrid CPU–GPU execution support in the skeleton programming framework SkePU. The Journal of Supercomputing (2019). DOI 10.1007/s11227-019-02824-7

[16] Phothilimthana, P.M., Ansel, J., Ragan-Kelley, J., Amarasinghe, S.: Portable performance on heterogeneous architectures. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pp. 431–444. ACM, New York, NY, USA (2013). DOI 10.1145/2451116.2451162

[17] Spampinato, D.G., Fabregat-Traver, D., Bientinesi, P., Püschel, M.: Program generation for small-scale linear algebra applications. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, pp. 327–339. ACM, New York, NY, USA (2018). DOI 10.1145/3168812

[18] Steuwer, M., Remmelg, T., Dubach, C.: Lift: A functional data-parallel IR for high-performance GPU code generation. In: Proc. CGO 2017, Austin, USA. IEEE (2017)

[19] Yasuhito Ogata, Toshio Endo, Naoya Maruyama, Satoshi Matsuoka: An efficient, model-based cpu-gpu heterogeneous fft library. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–10 (2008). DOI 10.1109/IPDPS.2008.4536163