International Journal of Parallel Programming (2023) 51:61–82
https://doi.org/10.1007/s10766-022-00746-1

Assessing Application Efficiency and Performance Portability in Single-Source Programming for Heterogeneous Parallel Systems

August Ernstsson[1] · **Dalvan Griebler**[2] · **Christoph Kessler**[1]

Received: 10 September 2022 / Accepted: 21 November 2022 / Published online: 6 December 2022

Abstract
We analyze the performance portability of the skeleton-based, single-source multi-backend high-level programming framework SkePU across multiple different CPU–GPU heterogeneous systems. Thereby, we provide a systematic application efficiency characterization of SkePU-generated code in comparison to equivalent hand-written code in more low-level parallel programming models such as OpenMP and CUDA. For this purpose, we contribute ports of the STREAM benchmark suite and of a part of the NAS Parallel Benchmark suite to SkePU. We show that for STREAM and the EP benchmark, SkePU regularly scores efficiency values above 80% and in particular for CPU systems, SkePU can outperform hand-written code.

Keywords Algorithmic skeletons · Parallel efficiency · Performance portability · Heterogeneous parallel computing · High-level parallel programming

1 Introduction

High-level parallel programming aims to simplify programming of systems with parallel (and possibly heterogeneous) hardware architectures. A high-level parallel programming model typically achieves this by abstracting away properties such as load balancing, synchronization, data movement, and other practical considerations,

✉ August Ernstsson
august.ernstsson@liu.se

Dalvan Griebler
dalvan.griebler@pucrs.br

Christoph Kessler
christoph.kessler@liu.se

1 PELAB, Department of Computer and Information Science, Linköping University, Linköping, Sweden

2 School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil

e.g., languages, compilers, and underlying APIs. Typically, the goal is also to provide *portability* across a large number of different platforms and even types of platforms (sometimes called "backends"). However, the efficiency of the resulting platform-specific code may vary considerably across the different platforms. This can be a particular challenge for single-source multi-backend high-level parallel programming models that need to generate, from the same high-level source, code for architecturally very diverse target platforms.

For high-level programming models, and in particular their concrete implementations in languages, libraries, and frameworks, it is therefore of interest to measure *performance portability* in addition to absolute performance. Performance portability is a property of a program and a set of specific target platforms; it quantifies the program's ability to run correctly and at decent efficiency across the given set of platforms without requiring (significant) modification of the source code. In order to achieve a good understanding of the overall performance portability, it is therefore important to find a representative platform set to gather experimental results from. The program(s) should also be representative of the target application area of the high-level parallel programming model, or alternatively, a general set of programs that can be used for comparison between a range of high-level parallel programming models or implementations.

A promising class of single-source high-level programming models are multi-backend *skeleton programming* frameworks such as SkePU [23], MueSLi [21], FastFlow [1], GrPPI [17] or SPar [25]. These frameworks provide a set of composable generic programming constructs (known as algorithmic skeletons) that implement certain parallelizable computation patterns, that can be parameterized in sequential, problem-specific code, and for which different platform-specific implementations (backends) are available.

This work investigates the performance portability of the skeleton-based, single-source multi-backend high-level data-parallel programming framework SkePU [23] across multiple different CPU–GPU heterogeneous systems. We provide a systematic application efficiency characterization of SkePU-generated code in comparison to equivalent hand-written code in more low-level parallel programming models such as OpenMP and CUDA. For this purpose, we contribute new SkePU ports of the STREAM benchmark suite and of a part of the NAS Parallel Benchmark suite.[1]

We show that for STREAM and the EP benchmark, SkePU regularly scores efficiency values above 80% and in particular for CPU systems, SkePU can outperform hand-written code. In addition, we provide code complexity metrics for the evaluated programs. It is shown that the size and complexity of SkePU code is significantly reduced compared to GPU implementations.

The remainder of this article is organized as follows: Sect. 2 presents background about high-level parallel programming, SkePU, performance portability and relevant benchmarks, esp. STREAM and NAS benchmarks. Related work is discussed in Sect. 3. Section 4 presents details about our experimental method. Results are listed

[1] The SkePU NPB implementations are available online at https://skepu.github.io/npb/.

and discussed in Sects. 5 and 6, respectively. Section 7 presents conclusions and Sect. 8 proposes future work.

2 Background

2.1 High-Level Parallel Programming and SkePU

Parallel programming is widely considered as being more difficult and error-prone than sequential programming, because the parallel execution dimension introduces new challenges, bug risks and potential performance issues that do not exist in the sequential computing world, such as load imbalance, race conditions, deadlocks and overheads for parallelism management, communication and synchronization.

Simple low-level extensions of sequential programming models by multithreading, accelerator control or message passing constructs, such as Pthreads, OpenCL or MPI respectively, leave the programmer alone with this additional complexity exposed. High-level parallel programming models promise to reduce complexity by providing structured programming constructs that manage parallelism, synchronization and communication for certain patterns of parallel computation. In particular, *skeleton programming* [11, 12] has been intensively researched during the last three decades, and improvements in programmability have been experimentally demonstrated [2, 3, 7, 15]. The approach is based on expressing computations in terms of pre-defined high-level constructs called *(algorithmic) skeletons* such as *map*, *reduce*, *scan* or *stencil*, which capture a specific, parallelizable computation pattern as a higher-order function that can be parameterized in user-provided code to instantiate executable code. All details of managing parallelism, communication and synchronization are encapsulated in the skeleton implementation. In this way, skeleton programs are, conceptually, no harder to write, read and maintain than well-structured sequential code for the same problem.

The reduced programming effort is usually paid for with some efficiency overheads compared to expert-written explicitly parallel code, and the skeleton approach is not applicable to computations that do not match any of the supported computation patterns. Nevertheless, the approach has been successfully demonstrated in research projects such as FastFlow [1], SkePU [23], SPar [25], and also been adopted in many modern parallel programming interfaces, such as Intel TBB, Nvidia Thrust, Hadoop MapReduce or Apache Spark.

Skeleton programming is particularly promising as a means to provide better code portability through a high-level abstraction which can more easily map to different types of target architectures (e.g., multicore CPU, GPU or cluster) in today's heterogeneous parallel computer systems. Even performance portability (see below) can benefit, as we shall see in this paper, as the programming system (compiler, runtime library) can build its own internal performance models for skeleton-based computations and is, in general, free to automatically select the expected fastest backend for each skeleton call.

In this work, we consider *SkePU*[2] [23] as a case study. SkePU is a domain-specific skeleton programming language embedded into modern C++. It provides currently 7 data-parallel variadic skeletons as well as STL-like generic abstractions for multidimensional operand data in memory and abstractions for different data access patterns. For each skeleton, implementations (*backends*) are provided for single-threaded and multi-threaded (OpenMP) CPU execution, single- and multi-GPU execution in CUDA and OpenCL, hybrid CPU–GPU execution, and cluster execution. The data abstractions for 1D, 2D, 3D and 4D generic array-based operands wrap array based data structures in main memory; they are referred to as "*smart*" *data-containers* as they transparently perform run-time optimizations such as coherent software caching and lazy device memory allocation and copying [13] and data locality optimizations [24]. SkePU is implemented by a light-weight source-to-source precompiler and an include-only runtime library for the interfaces and implementations of skeletons and data abstractions which makes intensive use of template metaprogramming in C++. SkePU is available as open-source with a permissive modified BSD license. Recent examples for the use of SkePU in HPC applications are described in [29].

2.2 Performance Portability

Performance portability is commonly understood as the ability of an application codebase, together with tools or layers of the hardware-software stack, to automatically achieve decent performance across different target architectures without significant changes. This is an intuitive quantity which is harder to define formally. In this work, we define performance portability in terms of *application efficiency* across platforms, which means the measured performance as a fraction of the best observed performance on the specific platform (by some other, ideally well optimized and tuned, code). This is contrasted with *architectural efficiency*, which instead compares the measured performance to the *peak hardware throughput* of the target system. Architectural efficiency is by definition lower than application efficiency.

Pennycook et al. [30] recently proposed a concrete metric for performance portability (PP) in terms of efficiency measurements $e_i(a,p)$ (for application a solving problem p on platform i) on a given specific set H of platforms:

$$\mathrm{PP}(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i\in H} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

[2] https://skepu.github.io/.

It is determined as the harmonic mean of the (application or architectural) efficiencies across all platforms in H, or 0 if any of these platforms does not support the computation (i.e., the program crashes or produces incorrect[3] results).

Notably, this metric is dependent on the considered platform set H and not an inherent property of an application. It makes sense to consider PP for architecturally related subsets of H, e.g., for the subset of all GPUs of interest, of all CPUs, as well as for all platforms together, because this will show common performance problems with certain architectural properties, as well as the sensitivity to product family variations (e.g., cache sizes). Taking the harmonic mean has a similar effect as minimization over the efficiencies for the different platforms, and also reflects the intuitive notion that adding new platforms to H will generally reduce the PP score, while algorithmic specialization (i.e., adding special code paths for some platforms) will generally increase the PP score.

We use this PP metric in this work, and time will tell if it achieves broad adoption in the scientific community.

2.3 Benchmarks

There is an ongoing effort to create SkePU implementations, and subsequently evaluations, of many benchmark workloads across several benchmark suites. Such suites include Rodinia [10], PARSEC [9] and its parallel derivate P3ARSEC [15], PolyBench [34], and NAS Parallel Benchmarks [5, 8, 27]. The complexity and effort required for benchmarking parallel programming models, interfaces, and frameworks is well-known [32] and examples of ongoing efforts to simplify and standardize parallel benchmark suites are many, including P3ARSEC and Task Bench. These efforts are seemingly conducted mostly in parallel to the work toward a widely accepted performance portability metric, and it remains one of the scientific goals of high-level parallel programming to merge these efforts into a methodology to evaluate programming models and frameworks across both application domains and platforms in a holistic process.

2.3.1 STREAM Benchmark Suite

The STREAM benchmark suite by McCalpin [28] of University of Virginia is primarily intended for measuring and comparing memory bandwidth of high-performance computing architectures. Versions of STREAM for distributed memory systems also exist, e.g. using MPI, but in this work we are working with single-node systems. However, heterogeneous architectures equipped with accelerators with separate memory spaces are also considered here.

[3] The notion of ”correct” behavior is not always obvious: especially when using accelerators or for more efficient parallelization, one might want to tolerate small differences in the result values within some limits, e.g. with respect to round-off errors of floating-point computations or the behavior of parallel pseudorandom number generation.

2.3.2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) was created and made available by the NASA Advanced Supercomputing division for benchmarking parallel hardware and software in the Computational Fluid Dynamics (CFD) application domain [8]. The benchmark suite is composed of five kernels (named Embarrassingly Parallel—EP, Multi Grid—MG, Conjugate Gradient—CG, Discrete 3D Fast Fourier Transform—FT, and Integer Sort—IS) and three pseudo-applications (named Block Tri-diagonal solver—BT, Scalar Penta-diagonal solver—SP, and Lower-Upper Gauss-Seidel solver—LU). They are well-known in the research community and represent recurrent linear algebra computations. The user can execute these programs with predefined workloads (named classes S, W, A, B, C, D, E, and F) that vary the computational problem's size. The original version was written in Fortran and the parallel implementations were in OpenMP and MPI. In recent years, an effort was made to provide parallel versions for C/C++ parallel programming frameworks on multicore systems [26, 27] as well as heterogeneous parallel programming on GPUs [5, 6, 19].

3 Related Work

Deakin et al. [16] provide ports of the STREAM benchmark set to several single-source parallel programming models: Kokkos, RAJA, OpenMP 4.x, OpenACC, SYCL, OpenCL and CUDA. (The "modern" version of SkePU that we use in our work did not yet exist at that time.) They evaluate performance portability for these programming models on a variety of GPU and CPU types from different vendors, including Intel Xeon Phi (Knights Landing). In contrast, our work focuses on SkePU and its supported platforms on both STREAM and NPB benchmarks. We also apply a stricter interpretation of performance portability, in that we do not admit any manual modifications to SkePU source code or backend code generated from the SkePU pre-compiler.

A number of papers such as [20] present and evaluate multi-platform implementations of the NPB, including GPU implementations in OpenCL and CUDA. A recent review and comparison of previous work on NPB parallelizations is given by Löff et al. [27]. There are also research efforts porting NPB to higher-level programming languages such as Python, providing parallel implementations for GPUs by using Numba (an LLVM-based Python JIT compiler) [18].

In the interest of brevity, we focus here on work based on single-source high-level programming models. Xu et al. [33] study the efficiency of the NAS Parallel Benchmarks rewritten in the directive-based single-source programming model OpenACC on GPUs and identify performance-critical GPU-specific optimizations such as array privatization that need be addressed by an OpenACC compiler. Performance is compared to hand-written OpenCL code for the NPB. A fundamental difference from our (NPB) implementations in SkePU is that SkePU is based on the more high-level skeleton concept rather than annotated sequential loop-based code as in OpenACC, so that SkePU's skeleton backend implementations are not constrained by the sequential code structure.

4 Methods

We compare an implementation of the STREAM benchmark suite in SkePU to the reference implementation across a set of 10 target *platforms*. Application *efficiencies* are calculated from the performance data, measured in terms of throughput datarate, and used as a basis for calculating the performance portability metric. The four STREAM workloads are evaluated on single and double precision floating-point data resulting in eight applications in total across ten platforms. We also document the programming effort required to implement the STREAM benchmarks in SkePU.

This work was conducted in three main steps: benchmark selection, platform selection, and performance evaluation.

4.1 Benchmark Selection and Implementation

The first step of this work was to select a set of benchmarks used to evaluate performance portability of SkePU. As the metric used takes efficiency data as input, a baseline requirement was to find data-parallel benchmark applications with independent reference-implementations available for all target platforms used in the evaluation. The choice fell on the STREAM benchmark suite as it is simple and well-known, facilitating the above requirement also with consideration of the limited scope of this project. SkePU has also not been evaluated on STREAM before, and a secondary aim is to further grow the set of benchmarks targeted by SkePU as part of the project.

In addition to the lightweight STREAM workloads, we also considered NPB as a means to select additional possible evaluation points, since these are different computations and are considered standard benchmarks for HPC evaluation. Like STREAM, NPB has not been subject to SkePU parallelization before, and given the recent work on NPB implementations in both C++ parallel CPU frameworks and CUDA for Nvidia GPUs [5, 27], we have good reference points for efficiency comparison against SkePU implementations. However, SkePUizing the entirety of NPB will be a future work, because the experience from this initial effort will indicate the viability of such a project. We therefore select two NPB kernels: *EP* and *CG*. EP shares similar properties to the STREAM kernels, such as being memory-bound, while the computational pattern modeled is not only a single Map, like STREAM, but rather a MapReduce, with a global reduction. One aspect that is also shared between STREAM and EP is that, during the entire program runtime, synchronization is only necessary at the start and end phases. In typical SkePU usage, we expect also to be able to handle multiple skeleton invocations in sequence in an efficient manner. To evaluate this, CG provides an iterative workload with each iteration also containing several global synchronization points. I.e., a SkePU implementation will have to consist of a substantial sequence of skeleton calls.

Table 1 Arithmetic intensity of individual STREAM kernels

Benchmark	Memory reads	Memory writes	Total mem accesses	Unique mem accesses	FLOPS
`copy`	1	1	2	2	0
`scale`	1	1	2	2	1
`add`	2	1	3	3	1
`triad`	2	1	3	3	2

While the STREAM reference implementation in C could be applied mostly unchanged, manual implementation work was required (as is explained in detail in Sect. 5.1) for SkePU versions of the workloads. Reference STREAM results for GPUs was based on open-source third-party implementations, but those required more tweaks to provide compatibility and a fair evaluation methodology. Table 1 lists the four workloads that make up STREAM: `copy`, `scale`, `add`, and `triad`, and shows the computational properties for each.

As NPB base-line to compare with, we use the CUDA implementations of EP and CG described in [27] as these have been experimentally shown [27] to perform on-par or better than other state-of-the art EP and CG implementations such as [20], see also Sect. 3.

4.2 Platform Selection

The platform selection is primarily guided by availability, but with the goal of having at least two physically distinct systems represented and also several different types of computational units. In this work, we are using the term **platform** in the sense of a *compilation target* + a *physical host system*, e.g., “sequential C++ processing on a server” or “laptop GPU with OpenCL runtime”.

The platforms are derived from three physical systems: laptop computer, the local server *Excess*, and the supercomputer cluster *Tetralith/Sigma*[4].

The laptop is equipped with a single 2 GHz quad-core *Intel Core i5* with *Intel Iris Plus* graphics and 16 GiB main memory and runs Mac OS. This GPU notably does not support double-precision compute kernels and cannot run CUDA programs.

Excess is a 12-core server (two six-core *Intel Xeon E5-2630L* CPUs with two-way hardware multi-threading, thus 24 logical cores) with one *Nvidia K20c* GPU and 64 GiB main memory. This system runs Ubuntu.

Tetralith and Sigma are large clusters with thousands and hundreds of nodes, respectively. We only use one node at a time in this work, of various configurations. Each Tetralith/Sigma node contains two Intel Xeon Gold 6130 CPUs for a total of 32 cores, with no hardware multi-threading. The minimum amount of node memory

[4] Tetralith and Sigma are sister clusters and share most of their hardware and software. Each cluster offers special nodes equipped, for example, with different GPU accelerators. We have used nodes from both clusters in this work.

Table 2 Platforms used in the performance portability evaluation

Platform name	System	Progr. model	Note
`excess-seq`	Excess	C/C++	
`excess-omp-12`	Excess	OpenMP	All physical cores
`excess-omp-24`	Excess	OpenMP	All logical cores
`excess-cl`	Excess	OpenCL	
`excess-cuda`	Excess	CUDA	
`laptop-seq`	Laptop	C/C++	
`laptop-omp-4`	Laptop	OpenMP	All physical cores
`laptop-omp-8`	Laptop	OpenMP	All logical cores
`laptop-cl-flush`	Laptop	OpenCL	
`laptop-cl-noflush`	Laptop	OpenCL	
`cluster`	Tetralith/Sigma	C/C++	
`cluster-omp-32`	Tetralith/Sigma	OpenMP	All physical cores
`cluster-v100-cuda`	Sigma	CUDA	
`cluster-t4-cuda`	Tetralith	CUDA	

is 96 GiB, with more available on GPU nodes. Tetralith GPU nodes are equipped with one Nvidia Tesla T4 GPU with 16 GiB of GPU memory, and the Sigma GPU nodes contain 4 Nvidia Tesla V100 SXM2 with 32 GiB of memory each.

From these systems we derive and define 14 target platforms (Table 2), using either different computational units or backend interfaces. Sequential platforms are targeted with C++ (OpenMP extensions disabled). For multi-core platforms we chose to run OpenMP with as many threads as there are physical and logical cores, respectively, but no thread pinning or similar was utilized.

On Excess, the GPU is targeted with both OpenCL and CUDA as separate platforms. The laptop GPU has a platform only for OpenCL, but for the sake of investigation, we define one "platform" as requiring a flush of GPU memory buffers between each measurement sample run, and one that only requires synchronizing with the GPU. This is purely a software differentiator but allows some insights into GPU bandwidth bottlenecks.

4.3 Evaluation Method

The STREAM and NPB benchmarks are compiled with GCC 10 and 11 in C++ mode (even for the reference program) with –O3 optimization level and no further optimization passes explicitly turned on. Evaluation is done using the default STREAM parameters: array sizes of 10 million elements and 10 runs per data point; and the default NPB problem size classes, except the D and E classes, as the time and memory requirements are impractically large.

Evaluation on STREAM benchmarks is repeated on single precision floating-point numbers and double-precision floating-point numbers, resulting in a memory load of 0.1 GB for the former and 0.2 GB for the latter.

5 Results

Results are reported in three evaluated categories: programming effort, performance, and performance portability.

5.1 Parallel Implementation in STREAM

The baseline code-churn of adapting the reference STREAM benchmarks to SkePU is very low. SkePU first needs one line minimum for its header inclusion: `#include <skepu>`.

5.1.1 Data Model

Next, the three arrays used in the workloads need to be wrapped in SkePU smart data-containers: SkePU cannot use raw arrays.

Listing 1: Smart data-container model SkePU-STREAM.

```
skepu::Vector<T> vec_a(&a[0], STREAM_ARRAY_SIZE+OFFSET, false);
skepu::Vector<T> vec_b(&b[0], STREAM_ARRAY_SIZE+OFFSET, false);
skepu::Vector<T> vec_c(&c[0], STREAM_ARRAY_SIZE+OFFSET, false);
```

The smart data-container definitions in Listing 1 utilize a SkePU smart data-container feature that claims "ownership" of raw memory regions and uses them internally for the lifetime of the data-container. This pointer is used as the memory buffer for sequential CPU or multi-threaded OpenMP backends; for GPUs additional device memory is allocated. The `false` argument passed here indicates that SkePU shall not deallocate the pointer as the lifetime of the container ends.[5]

5.1.2 Benchmarking Bookkeeping

SkePU skeleton invocations are not guaranteed to be synchronous. There are explicit optimizations in the framework relying on the opposite: that invocations are lazily evaluated only when strictly necessary. In fact, since STREAM is embarrassingly parallel, the iterated invocations are perfect targets for the tiling optimization on such lazily-built invocation chains. Even when this feature is explicitly disabled, e.g. the GPU backend implementations are also partly asynchronous, relying on internal GPU driver scheduling queues for synchronization of computation and requests for memory transfers. SkePU's interface exposes a `flush()` function (and related constructs) which guarantees a full synchronization to a completed skeleton invocation, but it is arguably too strong, as this operation will also trigger deallocation of GPU memory and possible memory transfers. The benchmarking code therefore uses an

[5] In fact, STREAM allocates these arrays on the stack, so freeing the pointers is always undefined behavior in C++.

internal SkePU synchronization operation between each measurement run to ensure as accurate results as possible.

Similarly, the result verification component of STREAM needs a flush call before it can run its checking algorithm.

5.1.3 Computations

Next, the skeletons representing the workload need to be defined. Each benchmark workload is straightforward to model with a single `Map` skeleton instance, and with a lambda expression this is done with a single statement/line-of-code per benchmark.

In Listing 2, `T` is a preprocessor macro symbol from the STREAM reference code defined to be either `float` or `double` at compile-time. While SkePU sometimes struggles with macro-heavy code, this usage here is handled properly by the precompiler.

Listing 2: STREAM kernels as SkePU skeletons.

```
auto skel_copy  = skepu::Map([](T a)
{
  return a;
});
auto skel_add   = skepu::Map([](T a, T b)
{
  return a + b;
});
auto skel_scale = skepu::Map<1>([](T a, T s)
{
  return a * s;
});
auto skel_triad = skepu::Map<2>([](T b, T c, T s)
{
  return b + s * c;
});
```

Note that the element-wise arity declared within chevrons is optional for `copy` and `add`, but mandatory for `scale` and `triad` as SkePU cannot otherwise distinguish the scalar `s` as not being an element-wise parameter.

5.2 Parallel Implementation of NPB-EP

The EP kernel requires relatively little effort to adapt for SkePU. A single `MapReduce` models the entire kernel, but the reduction step is more involved than the typical "dot-product" `MapReduce` archetype which reduces only a single value. EP computes global sums of `sx`, `sy`, and `q` values (see Listing 3), where the latter is a static array. SkePU can model this multi-way reduction pattern either by declaring a custom type and reducing on said type, or by using variadic tuple-return syntax introduced in SkePU 3 [23]. Notably, SkePU's internal reduction implementation for the OpenMP backend performs the final summation of thread-partial sums sequentially, while the reference OpenMP code [6] uses a mix of OpenMP pragma directives and a shared critical section, as shown in Listing 3.

Listing 3: Excerpt from the EP reference implementation in OpenMP.

```
#pragma omp parallel
{
    double t1, t2, t3, t4, x1, x2;
    int kk, i, ik, l;
    double qq[NQ];     /* private copy of q[0:NQ-1] */
    double x[NK_PLUS];

    for (i = 0; i < NQ; i++) qq[i] = 0.0;

    #pragma omp for reduction(+:sx,sy)
    for(k=1; k<=np; k++){
      int thread_id = omp_get_thread_num();
      /* ... business logic code omitted  */
    }

    #pragma omp critical
    {
        for (i = 0; i <= NQ - 1; i++) q[i] += qq[i];
    }
} /* end of parallel region */
```

5.3 Parallel Implementation of NPB-CG

NPB's CG kernel is considerably more complex than the other benchmark workloads considered in this paper. This is exemplified by the fact that CG cannot be implemented with just one SkePU skeleton call; rather twelve distinct skeleton instances are invoked in sequence for each CG iteration, totaling hundreds of skeleton calls during an entire CG execution. SkePU conceptually induces a global synchronization point between skeleton invocations, and while this can result in performance bottlenecks especially for iterative workloads, it is in most cases required by the CG algorithm as it contains global reductions and other non-local dependency structures. The skeleton structure of the SkePU implementation is closely following the CUDA kernels of the reference CUDA code,[6] which makes the GPU efficiency results of particular interest in this work.

In the process of porting the CUDA CG kernel to SkePU, we noted two limitations of the SkePU interface. Firstly, the current SkePU version does not include a smart data-container abstraction for sparse matrices. A sparse matrix can be modeled by a series of vector containers, but with additional programmer effort. Secondly, we observe a limitation in matrix-vector multiplication patterns as modeled by `Map` and SkePU's random-access container interface ("container proxies"). The typical way of encoding such computations in SkePU is shown in Listing 4.

[6] This is, in our experience, a good approach for SkePUizing applications with GPU implementations already available. For further information about programming in SkePU, we refer to Ernstsson [22].

Listing 4: Sparse matrix-vector multiplication in SkePU.

```
auto skepu_skel_three = skepu::Map([](
    skepu::Index1D     index,
    skepu::Vec<int>    colidx,
    skepu::Vec<int>    rowstr,
    skepu::Vec<double> a,
    skepu::Vec<double> p) -> double {
  int begin = rowstr(index.i);
  int end   = rowstr(index.i + 1);
  double sum = 0.0;
  for (int k = begin; k < end; ++k) {
    sum += a(k) * p(colidx(k));
  }
  return sum;
});

skepu_skel_three(q, colidx, rowstr, a, p);
```

Compared to the reference CUDA kernel in Listing 5, the SkePU variant parallelizes strictly on the elements in the output vector. The CUDA kernel allocates one block of threads for each output element, and can share the row calculations among the threads in the block, which in the SkePU case has to be managed by one thread. The reason for the discrepancy is that SkePU user functions are completely independent with no observable side effects, and as such the user functions executed within the same GPU block can neither communicate, synchronize, or even access the block size.

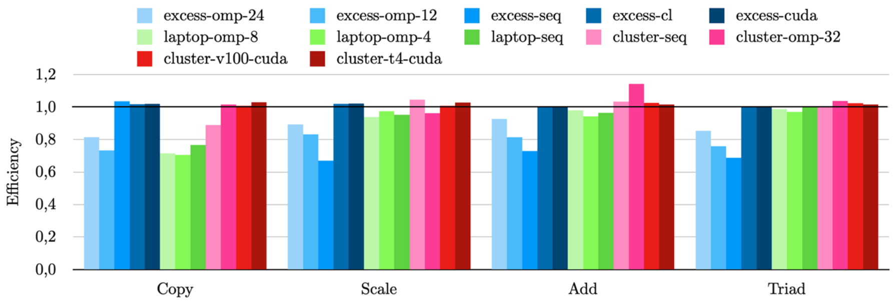


Fig. 1 Efficiency per platform on double-precision STREAM workloads

Listing 5: Sparse matrix-vector multiplication in CUDA.

```
__global__ void gpu_kernel_three(int colidx[],
    int rowstr[],
    double a[],
    double p[],
    double q[]) {
  double* share_data = (double*)extern_share_data;
  int j = (int) ((blockIdx.x*blockDim.x+threadIdx.x) / blockDim.x);
  int local_id = threadIdx.x;
  int begin = rowstr[j];
  int end   = rowstr[j+1];
  double sum = 0.0;
  for (int k=begin+local_id; k<end; k+=blockDim.x) {
    sum = sum + a[k]*p[colidx[k]];
  }
  share_data[local_id] = sum;

  __syncthreads();
  for (int i=blockDim.x/2; i>0; i>>=1) {
    if (local_id<i) { share_data[local_id]+=share_data[local_id+i]; }
    __syncthreads();
  }
  if (local_id==0){q[j]=share_data[0];}
}

gpu_kernel_three<<<blocks_per_grid_on_kernel_three,
    threads_per_block_on_kernel_three,
    size_shared_data_on_kernel_three>>>(
        colidx_device, rowstr_device,
        a_device, p_device, q_device);
```

With the exception of this parallelization scheme, all of the CUDA kernels are straightforward to adapt into SkePU skeletons while simplifying and reducing code size.

5.4 Performance Evaluation

The benchmark performance results for STREAM are presented in Figs. 1 and 2, visualizing the relative efficiency of the SkePU programs as compared to reference STREAM results. Note that the single-precision results contain two additional platforms: the laptop GPU is evaluated with and without explicit memory flushes between test runs.

Similarly, the performance results for NPB, EP in Fig. 3 and CG in Fig. 4, are presented as collected on the 12 platforms which can run double-precision calculations.

5.5 Performance Portability

We use *efficiency cascade plots* in Fig. 5 and Fig. 6 as proposed in [31] to visualize performance portability of the workloads when considering successively smaller platform sets. The rightmost data point in each line shows the PP metric for the entire set of platforms, and in each successive data point to the left, the least efficient platform is removed from the set and the PP metric is re-evaluated on the

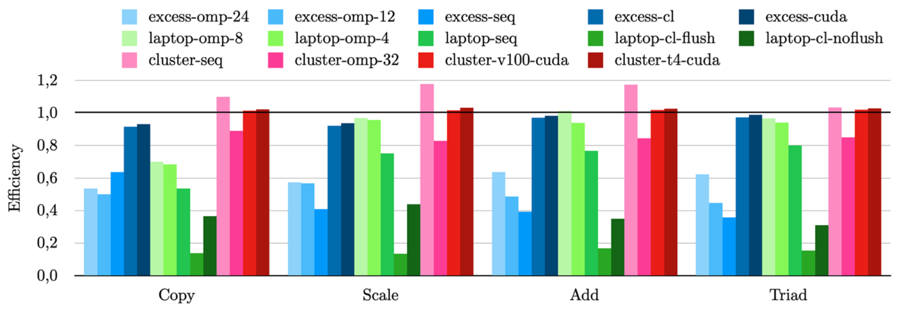


Fig. 2 Efficiency per platform on single-precision STREAM workloads

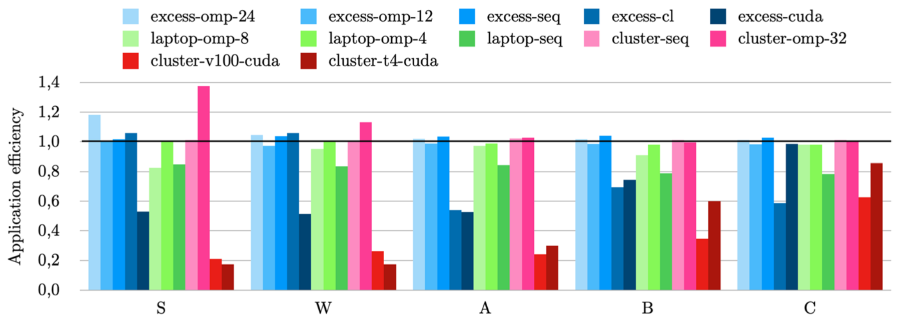


Fig. 3 Efficiency per platform and problem size class on NPB-EP

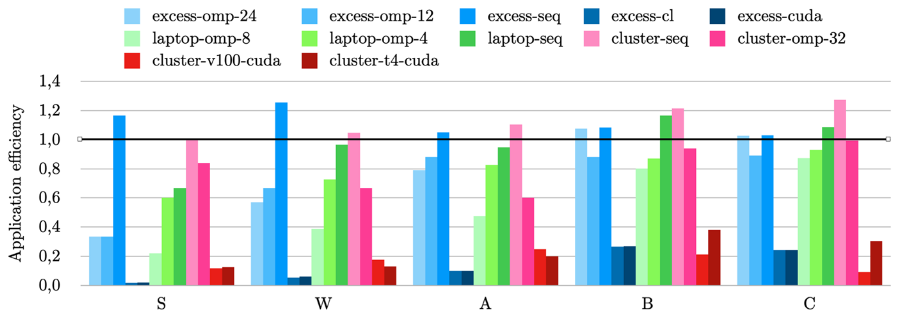


Fig. 4 Efficiency per platform and problem size class on NPB-CG

new subset. We use this method of visualization as the singular PP number is heavily influenced by the least efficient platform in the set. By computing the PP metric for the most interesting subsets of platforms, more detailed information can be conveyed.

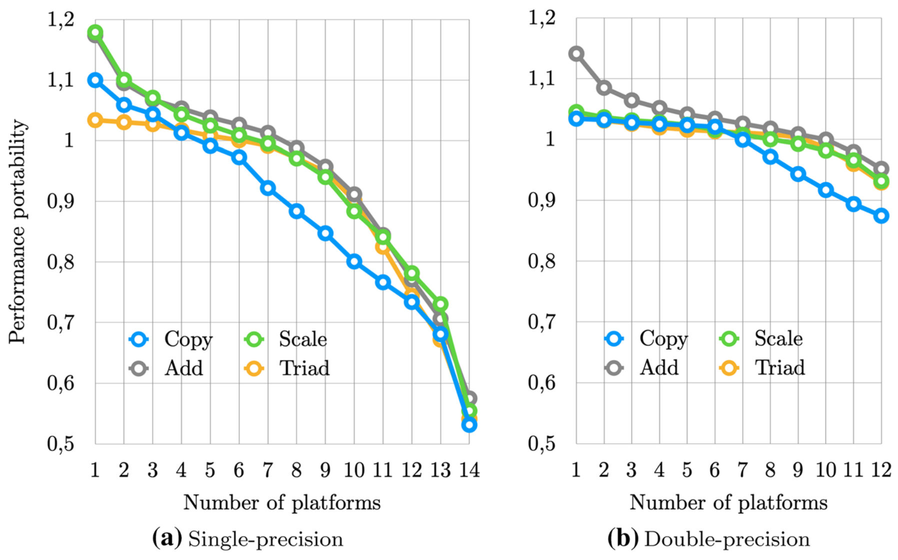


Fig. 5 Efficiency cascade plots for STREAM kernels by kernel and precision

Note that the order of platform removals may be different across workloads, so the different lines in each graph are not directly comparable.

This approach of visualization makes the different trade-offs between single- and double-precision workloads apparent. We see that the single-precision PP metrics are overall significantly lower, but the programs which depend on double-precision suffer from incompatibility on two platforms (and as such the PP metric would drop to zero on the rightmost platform subsets here).

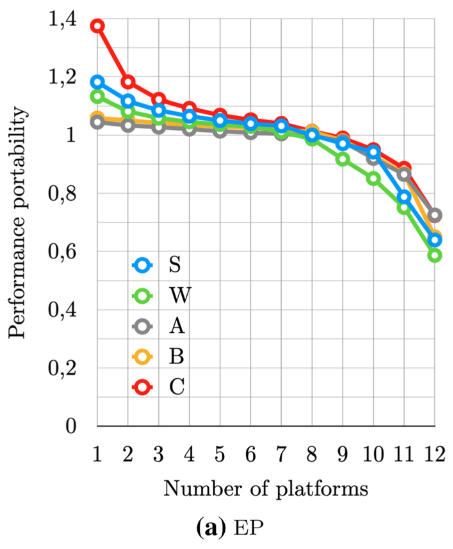
The efficiency cascade plots clearly show that the SkePU implementation of the `copy` workload exhibits worse performance portability than the other three workloads, which cluster tightly together in the plot. This fact is not as easy to discern in the traditional bar graphs.

5.6 Code Complexity Evaluation

In order to provide context into the single-source nature of SkePU code as compared to the separate code-bases of the reference implementations, we also conducted an evaluation of code complexity. Estimating programming effort for high-level programming models is an open research topic (see e.g., [4]), and for this paper, we limit the evaluation to established metrics not specific to parallel computing. We consider lines-of-code (NLOC), token count, and cyclomatic complexity. For this purpose, we use the Lizard tool.[7]

For the STREAM benchmark, we evaluate code complexity metrics on the Triad kernel (non-SkePU versions are taken from Deakin et al. [16]) and compare with the

[7] https://pypi.org/project/lizard/.



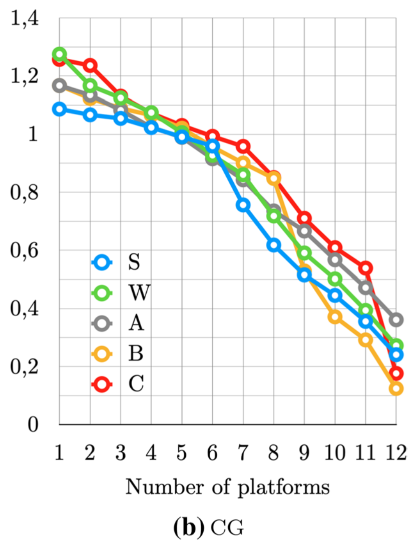


Fig. 6 Efficiency cascade plots for NPB kernels by problem size class

Table 3 Code complexity for STREAM benchmark `triad`

Code metric	Seq. C++	OpenMP	CUDA	OpenCL	SkePU (single-source)
NLOC	7	9	13	21	11
Token count	44	51	93	109	84
Cyclom. compl.	2	2	2	2	1

single-source SkePU code in Listing 6, see Table 3. The remaining STREAM kernels are highly similar in their computational structure, and therefore omitted from the evaluation.

Listing 6: SkePUized Triad kernel in the style of Deakin et al. [16].

```
template <class T>
void SkePUStream<T>::triad()
{
  const T scalar = 3.0;

  skepu::Vector<T> v_a(a), v_b(b), v_c(c);
  auto skel_triad = skepu::Map<2>([](T b, T c, T s)
  {
    return b + s * c;
  });
  skel_triad(v_a, v_b, v_c, scalar);
}
```

Table 4 Code complexity for NPB-EP and NPB-CG benchmarks

Benchm.	Code metric	Seq. C++	OpenMP	CUDA	SkePU (single-source)
NPB-EP	NLOC	163	178	298	233
	Token count	1273	1403	2111	1703
	Cyclom. compl.	30	36	38	30
NPB-CG	NLOC	547	561	1247	703
	Token count	3517	3428	6522	4769
	Cyclom. compl.	109	106	216	136

For the NPB benchmarks, we run the reference benchmark implementations of Araujo et al. [6] directly through the evaluation tool. The evaluation therefore includes benchmark overhead code, which is equal across data points. Note, in particular, that the GPU-enabled variants (CUDA and SkePU) include per-kernel profiling code, which makes them slightly longer. The results are summarized in Table 4.

6 Discussion

From the STREAM results, we observe that the SkePU programs with single-precision workloads are less efficient than the corresponding double-precision ones. The STREAM reference implementation is a set of microbenchmarks in a tightly controlled environment: program parameters are all compile-time configuration options, including array size. The arrays are allocated on the program stack, so all program addresses are statically known, allowing for far-reaching optimization opportunities by the compiler. In comparison, SkePU skeleton code is generated behind layers of abstractions. SkePU smart data-containers operate on pointers that are normally dynamic allocations of arbitrary size, and the backend selection is entirely a run-time decision. This means that even though SkePU is embedded into the STREAM reference codebase with only minor alterations, the static information is unlikely to propagate all the way down to the skeleton internals unless the backend compiler is extremely aggressive with global static analysis. Examples of optimization stages that can be affected include inlining of SkePU user function calls, loop unrolling, loop fusion, auto-vectorization, and pointer de-aliasing.

The STREAM workloads are all memory-bound, so the hyper-threaded platforms do not scale linearly in throughput from the baseline platforms with thread counts matching the number of physical cores. However, the laptop still benefits quite a bit from hyper-threading, while Excess is largely flat.

The application efficiency numbers presented in the paper could be made more accurate if per-system-tuned reference implementations were used. Performance efficiency properties differ a lot between the evaluated systems, even for the same types of backend targets, and several times the efficiency is recorded as over 100%. This indicates that the task of finding optimal reference implementations requires implementations with built-in platform-tuning capabilities. In particular, the EP kernel frequently results in cases where the SkePU version outperforms the reference code. For OpenMP, it is likely that the synchronization approach by SkePU contributes to this, as discussed in Sect. 5.2.

For the CG kernel, the efficiency results on GPU platforms are very low. For smaller problem sizes, the skeleton invocation overhead is contributing to these inefficiencies, but more importantly SkePU programs induce initialization delays due to environment set-up and lazy memory allocations on device, and so on. These delays may still occur in hand-written implementations, but the abstractions in SkePU make it impossible to assure that initialization delays happen outside of the critical timing regions, without considerable code instrumentation. For larger problem sizes, such overhead is a smaller quota of the total execution time. For large CG sizes, instead, the issue discussed in Sect. 5.3 becomes important. The matrix-vector multiplication phase becomes dominating at large problem sizes, and the parallelization inefficiency in the SkePU version becomes a performance bottleneck.

From the code complexity evaluation, we see that SkePU variants of the benchmark kernels are generally somewhat longer and more complex than serial reference code, but considerably shorter than GPU-parallelized implementations. Note also that the single-source SkePU code can run on all tested platforms without further modification.

7 Conclusions

Of the benchmarks evaluated in this work, we have shown that STREAM benchmarks, specifically in their original double-precision form, and the EP kernel in NPB result in high efficiency and performance portability when implemented using the SkePU skeletons and smart data-containers. For the CG benchmark, the results are not as consistent and SkePU's efficiency is highly platform-dependent. We have identified specific performance bottlenecks in SkePU, in particular for GPU backends.

The results demonstrate that SkePU-parallelized programs can achieve good application efficiency and even outperform hand-written parallel code in some cases, even though SkePU code is single-source and often shorter than any of the individual platform-specific programming models. Note also that this work does not even take the auto-tuning functionality of SkePU [14] into account, which does provide an advantage for SkePU compared to distinct and unrelated parallel implementations.

8 Future Work

In the long term, we aim for complete SkePU coverage of the NPB kernels. In this work, we have worked entirely within the existing pattern and feature set of SkePU, but some of the kernels do not fit well the data-parallel patterns that are available, and may be suitable case studies for extensions to the framework. One such possible extension is a multi-backend sorting skeleton. Such a construct would be useful for a convenient and efficient implementation of the IS kernel in NPB.

In addition to new benchmarks, the evaluation can be extended by considering further platforms, in particular multi-node cluster platforms and multi-GPU as well as CPU+GPU hybrid computing.

A further direction for future work is to extend our performance portability analysis of SkePU by including other single-source high-level parallel programming models, such as directive based models like OpenACC, SPar, as well as other single-source skeleton programming frameworks.

Acknowledgements This work has been partly funded by ELLIIT, project GPAI. We thank SNIC and NSC for providing access to HPC resources Tetralith (SNIC 2021/22-971) and Sigma (LiU-gpu-2021-1) that were used for this work.

Author contributions A.E. wrote the initial manuscript and conducted the experiments. D.G. and C.K. contributed considerably to the manuscript and reviewed the experimental results. All authors reviewed the manuscript.

Funding Open access funding provided by Linköping University.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-Level and Efficient Streaming on Multicore, chapter 13, pp. 261–280. Wiley, Hoboken (2017)
2. Andrade, G., Griebler, D., Santos, R., Danelutto, M., Fernandes, L.G.: Assessing coding metrics for parallel programming of stream processing programs on multi-cores. In: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 291–295, (2021)
3. Andrade, G., Griebler, D., Santos, R., Fernandes, L.G.: A parallel programming assessment for stream processing applications on multi-core systems. Comput. Stand. Interfaces **84**, 103691 (2022)
4. Andrade, G., Griebler, D., Santos, R., Kessler, C., Ernstsson, A., Fernandes, L.G.: Analyzing programming effort model accuracy of high-level parallel programs for stream processing. In: 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 1–4 (2022)
5. Araujo, G., Griebler, D., Rockenbach, D.A., Danelutto, M., Fernandes, L.G.: NAS parallel benchmarks with CUDA and beyond. Softw. Pract. Exp. 1–28 (2021)
6. Alves de Araujo, G., Griebler, D., Danelutto, M., Fernandes, L.G.: Efficient NAS parallel benchmark kernels with CUDA. In: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 9–16 (2020)

7. Arvanitou, M., Ampatzoglou, A., Nikolaidis, N., Tzintzira, A., Ampatzoglou, A., Chatzigeorgiou, A.: Investigating trade offs between portability, performance and maintainability in exascale systems. In: 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 59–63 (2020)
8. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D. Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks-summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing'91, pp. 158–165, New York, NY, USA. Association for Computing Machinery (1991)
9. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT'08, pp. 72–81, New York, NY, USA. Association for Computing Machinery, (2008)
10. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC) pp. 44–54 (2009)
11. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004)
12. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman and MIT Press, Cambridge (1989)
13. Dastgeer, U., Kessler, C.: Smart containers and skeleton programming for GPU-based systems. Int. J. Parallel Prog. **44**(3), 506–530 (2016)
14. Dastgeer, U., Li, L., Christoph, K.: Adaptive implementation selection in the SkePU skeleton programming library. In: Revised Selected Papers of the 10th International Symposium on Advanced Parallel Processing Technologies---Volume 8299, APPT 2013, pp. 170–183. Springer, Berlin (2013)
15. De Sensi, D., De Matteis, T., Torquati, M., Mencagli, G., Danelutto, M.: Bringing parallel patterns out of the corner: The P3ARSEC benchmark suite. ACM Trans. Archit. Code Optim. **14**(4), 1–26 (2017)
16. Deakin, T., James, P., Matt, M., Simon, M.S.: GPU-stream v2.0: benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds) High Performance Computing, pp. 489–507. Springer, Cham (2016)
17. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurr. Comput. Pract. Exp. **29**(24), 4175 (2017)
18. Domenico, D.D., Cavalheiro, G.G.H., Lima, J.V.F.: Nas parallel benchmark kernels with python: a performance and programming effort analysis focusing on GPUs. In: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP) pp. 26–33 (2022)
19. Do, Y., Kim, H., Oh, P., Park, D., Lee, J.: SNU-NPB 2019: parallelizing and optimizing NPB in OpenCL and CUDA for modern GPUs. In: 2019 IEEE International Symposium on Workload Characterization (IISWC), pp. 93–105, IEEE (2019)
20. Do, Y., Kim, H., Oh, P., Park, D., Lee, J.: SNU-NPB 2019: parallelizing and optimizing NPB in OpenCL and CUDA for modern GPUs. In: International Symposium on Workload Characterization (IISWC), pp. 93–105 (2019)
21. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. High Perform. Comput. Netw. 7(2):129-138 (2012)
22. Ernstsson, A.: Pattern-based Programming Abstractions for Heterogeneous Parallel Computing. PhD thesis, Linköping University Electronic Press (2022)
23. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: SkePU 3: portable high-level programming of heterogeneous systems and HPC clusters. Int. J. Parallel Prog. **49**, 846–866 (2021)
24. Ernstsson, A., Kessler, C.: Extending smart containers for data locality-aware skeleton programming. Concurr. Comput. Pract. Exp. **31**(5), e5003 (2019)
25. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: SPar: a DSL for high-level and productive stream parallelism. Parallel Process. Lett. 27(01):1740005 (2017)
26. Griebler, D., Löff, J., Mencagli, G., Danelutto, M., Fernandes, L.G.: Efficient NAS benchmark kernels with C++ parallel programming. In: 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 733–740 (2018)
27. Löff, J., Griebler, D., Mencagli, G., Araujo, G., Torquati, M., Danelutto, M., Fernandes, L.G.: The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. Future Gener. Comput. Syst. **125**, 743–757 (2021)

28. McCalpin, J.D.: STREAM benchmark (1995)
29. Papadopoulos, L., Soudris, D., Kessler, C., Ernstsson, A., Ahlqvist, J., Vasilas, N., Papadopoulos, A.I., Seferlis, P., Prouveur, C., Haefele, M., Thibault, S., Salamanis, A., Ioakimidis, T., Kehagias, D.: Exa2pro: a framework for high development productivity on heterogeneous computing systems. IEEE Trans. Parallel Distrib. Syst. **33**(4), 792–804 (2022)
30. Pennycook, S.J., Sewall, J.D., Lee, V.W.: Implications of a metric for performance portability. Future Gener. Comput. Syst. **92**, 947–958 (2019)
31. Sewall, J., Pennycook, S.J., Jacobsen, D., Deakin, T., McIntosh-Smith, S.: Interpreting and visualizing performance portability metrics. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 14–24 (2020)
32. Slaughter, E., Wu, W., Fu, Y., Brandenburg, L., Garcia, N., Kautz, W., Marx, E., Morris, K.S., Cao, Q., Bosilca, G., Mirchandaney, S., Leek, W., Treichlerk, S., McCormick, P., Aiken, A.: Task bench: a parameterized benchmark for evaluating parallel runtime performance. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–15 (2020)
33. Xu, R., Tian, X., Chandrasekaran, S., Yuan, Y., Chapman, B.: NAS parallel benchmarks for GPGPUs using a directive-based programming model. In: Proceedings of the LCPC 2014, LNCS 8967, pp. 67–81. Springer, Berlin (2015)
34. Yuki, T., Pouchet, L.-N.: Polybench 4.0 (2015)