Energy Efficient Stencil Computations on the Low-Power Manycore MPPA-256 Processor*

Emmanuel Podestá Jr., Bruno Marques do Nascimento, and Márcio Castro[0000−0002−9992−8540]

Graduate Program in Computer Science (PPGCC)
Federal University of Santa Catarina (UFSC)
Florianópolis, SC, Brazil
{emmanuel.podesta, bruno.mn}@grad.ufsc.br, marcio.castro@ufsc.br

Abstract. A new class of highly-parallel low-power manycore chips that cope with energy constraints have been unveiled. Sunway's SW26010 and Kalray's MPPA-256 are examples of them, featuring more than two hundred cores in a single low-power chip. Although they may present better energy efficiency than general-purpose multicore processors, architectural characteristics such as their limited amount of distributed on-chip memory make the development of efficient scientific parallel applications a challenging task. In this paper we propose and evaluate a new back-end of PSkel, a framework that provides a single high-level abstraction for stencil programming on CPUs and GPUs, for the low-power manycore MPPA-256 processor. This relieves programmers of the burden of explicitly dealing with communications and the hybrid underlying programming model of MPPA-256. Our results showed that the energy consumption of stencil applications running on MPPA-256 is up to 7.34x and 4.71x lower than on an Intel Xeon E5 multicore and NVIDIA Tesla K40 GPU, respectively.

Keywords: MPPA-256 · Manycore · PSkel · Energy efficiency.

1 Introduction

High Performance Computing (HPC) platforms have been evaluated based almost exclusively on their raw processing speed. However, their energy efficiency have become as important as raw performance. Because of that, a new class of highly-parallel low-power manycore chips that cope with energy constraints was unveiled. Sunway's SW26010 [6] and Kalray's MPPA-256 [5] are examples of such processors, providing more than two hundred low-power autonomous cores that can be exploited through both data and task parallelism.

* The authors would like to thank CAPES and CNPq for funding this research. This work was also supported by STIC-AmSud/CAPES scientific cooperation program under EnergySFE research project grant No. 99999.007556/2015-02. Finally, a special thank to Jean-François Méhaut (LIG/CNRS) for giving access to the MPPA-256 platform to the authors.

Although low-power manycores may present better energy efficiency than general-purpose multicore processors [5], their particular architectural characteristics make the development of efficient scientific parallel applications a very challenging task [2, 18]. Processing cores with non-coherent caches are usually distributed in a clustered architecture that features a hybrid programming model. On the one hand, cores in the same cluster share a limited amount of directly addressable memory. On the other hand, distinct clusters must communicate through the Network-on-Chip (NoC) in a distributed fashion. For that reason, communication costs between cores may vary significantly, depending on the location of the communicating cores on the NoC.

One possible approach to ease the development of parallel applications for low-power manycores is through the use of skeletons [3]. Skeletons allow programmers to focus on designing algorithms rather than worrying about synchronization issues and task scheduling, which are transparently handled by the skeleton framework, thereby speeding up application development and debugging. Among several existing patterns of parallel skeletons (*e.g.*, map, reduce, pipeline and scan), the stencil pattern has been used in applications of many important fields, such as quantum physics, weather forecasting and digital image processing [8]. The stencil pattern operates on n-dimensional data structures, using an input data value and its neighbors to compute the corresponding output data element. This process is repeated for every input data value in the n-dimensional data structure.

Indeed, many frameworks have been proposed to ease the development of parallel stencil computations on multicores and Graphics Processing Units (GPUs) such as PSkel [11], SkePU [16] and SkelCL [15]. In particular, PSkel is a stencil framework that provides a single high-level abstraction for stencil programming on heterogeneous CPU-GPU systems, while allowing automatic data partition, assignment and computation to both CPU and GPU. In this paper we present the design, implementation and evaluation of a new back-end of PSkel for the low-power manycore MPPA-256 processor (PSkel-MPPA). The same high-level, low overhead and intuitive PSkel code that already ran transparently on GPUs and multicores is extended to run also on the MPPA-256 architecture. This relieves programmers of the burden of explicitly dealing with NoC communications, the hybrid underlying programming model and the absence of cache coherence on MPPA-256. Our solution uses a trapezoidal tiling technique to reduce the number of communications and synchronization barriers on MPPA-256, which improves considerably the overall performance. Our results show that the energy consumption of stencil applications on MPPA-256 is up to 7.34x and 4.71x lower than on an Intel Xeon E5 multicore and NVIDIA Tesla K40 GPU, respectively, while presenting competitive performance.

The remainder of this paper is organized as follows. Section 2 presents an overview of MPPA-256 and PSkel. Next, Section 3 describes our proposal (PSkel-MPPA) as well as its implementation details. Then, Section 4 presents the results obtained with PSkel-MPPA, comparing them against reference implementations

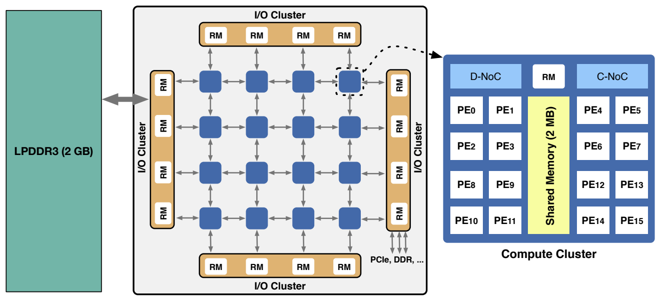


Fig. 1. Overview of the low-power MPPA-256 manycore processor.

of PSkel for multicores and GPUs. Section 5 discusses related work. Finally, Section 6 concludes this paper.

2 Background

2.1 MPPA-256

MPPA-256 is a single-chip low-power manycore processor developed by Kalray that integrates 256 user cores and 32 system cores in 28 nm CMOS technology running at 400 MHz. These cores are distributed across 16 compute clusters and 4 Input/Output (I/O) clusters that communicate through data and control NoCs. The board used in this paper has one of the I/O clusters connected to an external Low-Power Double Data Rate 3 (LPDDR3) of 2 GB. Fig. 1 shows an architectural overview of the MPPA-256 processor.

Overall, each compute cluster has the following components:

– 16 cores called Processing Elements (PEs), which are dedicated to run user threads (one thread per PE) in non-interruptible and non-preemptible mode. Each PE has private 2-way associative 32 kB instruction and data caches;
– One Resource Manager (RM), which is responsible for running the operating system and managing communications;
– A low-latency shared memory of 2 MB, which enables a high bandwidth and throughput between PEs within the same compute cluster; and
– Two NoC controllers, one for data and other for control.

The processor features a distributed memory model. Compute clusters and the I/O clusters have their own address spaces. Applications must use two parallel programming libraries to exploit all processor resources: a thread library (Pthread or OpenMP) and a proprietary library called Asynchronous Operations Application Programming Interface (Async API). The former is used to parallelize computations in computing clusters via shared memory. The latter follows a distributed memory model and must be used for cluster-cluster and cluster-I/O communications through the NoC.

Async API is based on one-sided communications between the compute clusters' local memory and LPDDR3. The main concepts behind Async API are *execution domains*, *segments* and `put/get` operations. The execution domain represents a set of cores sharing a local memory, being isolated from other execution domains. Considering the MPPA-256 distributed memory model, an execution domain corresponds to a compute cluster or to an I/O cluster. Memory that is not directly accessible from the cores of an execution domain can be structured into segments, which correspond to the entire or part of the local memory of cores located in another execution domain. Each segment has a *unique signature*, which is specified when the segment is created in an execution domain through the `mppa_async_segment_create()` function. Then, other execution domains can reference a previously created segment by passing its unique signature the function `mppa_async_segment_clone()`. Once segments are created and referenced by different execution domains, one should use put/get operations to read data from a remote segment into the local memory (`get` operation) or to write local data to the remote segment (`put` operation). Different flavors of these operations are available in Async API, allowing contiguous or spaced data transfers (*e.g.*, `mppa_async_put()` and `mppa_async_get_spaced()`) as well as 2D block transfers (`mppa_async_sget_block2d()`), which is useful for transferring 2D data blocks.

The execution flow of an MPPA-256 application is the following. The main process (called *master process*) runs on an RM of the I/O cluster connected to the LPDDR3 and is responsible for allocating the input data in its local memory (LPDDR3) and spawning *worker processes* (one for each compute cluster) by calling the `mppa_power_base_spawn()` function. The necessary data segments should be created by the master process so it can exchange data with the compute clusters. Finally, the master process should wait all worker processes to finish by calling the `mppa_power_base_waitpid()` function. Each worker process should make references to remote segments allocated in the LPDDR3 to exchange data during the execution and may create up to 16 threads using Pthread or OpenMP (one thread for each PE) to perform computations in parallel. Each PE has its own private cache memory without any automatic coherence mechanism among the remaining PEs cache memories. Although this improves the cache performance, it requires the developer to explicitly flush data when needed.

2.2 Stencil Pattern and PSkel

The stencil computational pattern operates on n-dimensional data structures and uses a sliding window (*a.k.a* mask) that scans the entire input data set and produces output data using a user-defined stencil kernel function. The mask size corresponds to a specific number of neighbors of each element of the input data. The stencil application repeats that process on every element of the input data. Stencil applications can be iterative, which means that the output data produced after an iteration t is used as the input for an iteration $t + 1$.

PSkel is a framework for high-level programming stencil computations, based on the concept of parallel skeletons, which offers parallel execution support on

```
 1 __parallel__ void stencilKernel(Array2D<float> A, Array2D<float> B,
 2                               struct Arguments args, int x, int y) {
 3   B(x,y) = args.alpha*(A(x,y+1)+A(x,y-1)+A(x+1,y)+A(x-1,y)+args.beta);
 4 }
 5
 6 void jacobi(float *A, float *B, int M, int N, float alpha, float beta,
 7             int timesteps) {
 8   Array2D<float> input(A,M,N);
 9   Array2D<float> output(B,M,N);
10   struct Arguments args(alpha, beta);
11   Stencil2D<Array2D<float>, struct Arguments> stencil(input,output,args);
12   stencil.runIterativeGPU(timesteps);
13 }
```

Fig. 2. Simplified example of a PSkel stencil code.

CPUs and GPUs [11]. PSkel offers a single programming interface, decoupled from the runtime back-ends, that releases the programmer from the responsibility of writing boiler-plate code for parallel stencil computation. Instead, the programmer is responsible for implementing a stencil kernel describing solely the computation, while the framework translates the abstractions described into low-level parallel C++ code. Synchronization, memory management and data transfers are transparently handled by the framework.

Fig. 2 shows an example of the Jacobi method for solving matrix equations [4] written in PSkel. The PSkel Application Programming Interface (API) provides templates for manipulating input and output data via template classes for n-dimensional arrays, called `Array`, `Array2D` (Fig. 2, lines 8–9) and `Array3D`. These abstractions provide methods that encapsulate the data management procedures, such as memory allocation, memory copy and data transfer (*e.g.*, communication between CPU and GPU). Moreover, it provides abstractions for specifying the stencil kernel and to manage the stencil execution. The stencil kernel (prototype function `stencilKernel()`) is the application specific method that describes the computation that will be performed on each entry of the input array and its neighbors (Fig. 2, lines 1–4). The `stencilKernel()` prototype function must be implemented by the user of the PSkel. Finally, the API provides a set of classes for managing the whole execution of the user-defined number of iterations of the stencil kernel over the input and output data, such as `Stencil`, `Stencil2D` (Fig. 2, line 11) and `Stencil3D`.

In the given example, the `stencilKernel()` function will be executed on the GPU. The `runIterativeGPU()` method hides from the user all the CUDA code needed to correctly execute the specified stencil kernel on the GPU.

3 PSkel-MPPA

As previously discussed in Section 2.2, PSkel currently supports the execution of stencil applications on CPUs and GPUs. In this paper we propose a new back-end of PSkel for the low-power manycore MPPA-256 processor, which differs significantly from the CPU and GPU ones due to the intrinsic characteristics of MPPA-256 discussed in Section 2.1, such as: (i) limited amount of on-chip memory; (ii) clustered architecture with NoC constraints; (iii) processing cores with non-coherent caches; and (iv) proprietary low-level communication API.

The new back-end, named PSkel-MPPA, supports 2D stencils (`Stencil2D` class in PSkel) and adopts the master-worker model. The master process is executed in the I/O cluster connected to the LPDDR3 memory, in which the input and output data (`Array2D` objects) are allocated, whereas the worker processes are executed on the compute clusters (one worker process per compute cluster) to perform the stencil computation in parallel. Given the memory limitation inside compute clusters (2 MB), the input `Array2D` is partitioned into *tiles* of fixed user-defined size to be sent to them. When tiling stencil computations, neighborhood dependencies inherent to the stencil parallel pattern must be considered before partitioning the input data.

We used the trapezoidal tiling technique to handle neighborhood dependencies in PSkel-MPPA, resulting in redundant data and computation per tile [13]. We use a formal definition to illustrate this technique. Let A be a 2D data matrix, with dimensions $\dim(A) = (w, h)$, where w and h are, respectively, its width and height. Using tiles of dimensions (w', h') yields $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil$ possible tiles of A. Let $A_{i,j}$ be one such tile, where $0 \leq i < \lceil \frac{w}{w'} \rceil$ and $0 \leq j < \lceil \frac{h}{h'} \rceil$. $A_{i,j}$ has offset (iw', jh') relative to the top left corner of A and $\dim(A_{i,j}) = (\min\{w', w - iw'\}, \min\{h', h - jh'\})$. The offset is an indexing displacement required for accessing the elements of the tile.

Fig. 3 shows a graphical view of this technique. A logical tile (inner solid line) is contained in a 2D data matrix (outer dashed line) with vertical and horizontal offsets given by jh' and iw'. If t iterations of a stencil application should be executed, it is possible to compute t' consecutive iterations on $A_{i,j}$ ($t' \in [1, t]$) without the need of any data exchange between adjacent tiles (*a.k.a* inner iterations). To do so, the logical tile ($A_{i,j}$) must be enlarged with a ghost zone (area between the inner solid line and the outer solid line), which is comprised of a halo region (the area between the inner solid line and the inner dashed line). Let r be the most distant displacement required for the neighborhood defined by the stencil mask. The area of range r comprising the neighborhood is denominated *halo region*. The number of adjacent halo regions that compose the ghost zone is proportional to t'. Thus, the enlarged tile $A^*_{i,j}$ has offsets $(\max\{iw' - rt', 0\}, \max\{jh' - rt', 0\})$ relative to A. Thus, sizing the ghost zones poses a trade-off between the cost of redundant computations and the reduction in communication and synchronizations on the NoC when processing iterative stencil computations on MPPA-256.

The execution flow of PSkel-MPPA follows the one described in Section 2.1. During the initialization phase, the master process running on the I/O cluster allocates the input and output data in the LPDDR3, and creates a specific segment for each one of them. Next, it calculates the number of enlarged tiles that will be produced as well as their dimensions based on: i) user-defined parameters, such as the input data and logical tile dimensions, the number of compute clusters and the number of inner iterations; and ii) stencil kernel parameters, such

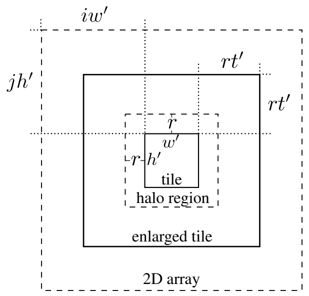


Fig. 3. 2D tiling [13].

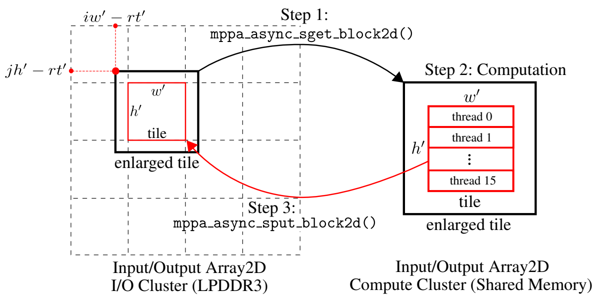


Fig. 4. Communications with `block2d`.

as the mask size. Then, it spawns up to 16 worker processes (one for each compute cluster) and informs each worker process about the number of enlarged tiles produced, their dimensions and the subset of tiles it should compute later on. Finally, the master process waits for all workers to finish. Each worker process, on the other hand, allocates data to store the input and output enlarged tiles in the compute cluster local memory and clones both input and output remote segments that were already created by the master process to make data transfers further on. The initialization phase in both master and worker processes is encapsulated in the `Stencil2D` class.

The computation phase consists of the execution of the stencil kernel by the worker processes. The following three main steps are performed to compute each tile assigned to a worker process: 1) the enlarged tile is extracted from the input data allocated in LPDDR3 and transferred to compute cluster local memory to be processed; 2) t' iterations of the stencil kernel (inner iterations) are executed by the worker process over the enlarged tile; and 3) the resulting logical tile is transferred back from the compute cluster local memory to its corresponding position in the LPDDR3. Once all tiles assigned to each worker process were successfully computed, all worker processes must synchronize at a global barrier, since the data computed during t' iterations will be needed by the others in the following iteration to solve neighborhood dependencies. We used the `mppa_rpc_barrier_all()` function for this purpose. The whole procedure described before is then repeated until the total number of iterations defined by the user (t) is reached.

The aforementioned steps are depicted in Fig. 4 and they are described in more detail below:

Step 1. Based on the information given by the master process during the spawn procedure, the worker process is capable of calculating the coordinates of each enlarged tile assigned to it with respect to the input data allocated in the LPDDR3 ($iw' - rt'$ and $jh' - rt'$ coordinates) without any other intervention from the master process. The `mppa_async_sget_block2d()` function takes such information and the block size as input parameters and it transfers the enlarged tile to be processed by the worker process from the input remote segment into the compute cluster local memory through the NoC.

Step 2. The worker process computes t' iterations of the user-defined stencil kernel over the enlarged tile. In each t' iteration, the computation is parallelized by means of an OpenMP parallel region. The parallel region creates up to 16 threads (one for each PE). Each PE is responsible for executing the stencil kernel on a subset of the enlarged tile cells.

Step 3. After the stencil kernel computation, the resulting logical tile is transferred back to the LPDDR3. The `mppa_async_sput_block2d()` function is used for this purpose, allowing the logical tile to be extracted from the enlarged tile in the compute cluster local memory and transferred to its corresponding position in the output remote segment.

Fortunately, all complex tasks related to the tiling technique, NoC communications and adaptations discussed in this section are hidden from the developers, since they were included in the back-end of PSkel. This means that current applications developed with the PSkel framework can run seamlessly on MPPA-256 without any source code modifications.

4 Experimental Evaluation

4.1 Platforms, Applications and Inputs

We evaluate the performance and energy consumption of the proposed solution (PSkel-MPPA) against reference multicore and GPU implementations available in PSkel. Energy measurements were collected from power and energy sensors available on MPPA-256, which include all clusters, memory (on-chip memory and LPDDR3) and NoCs. The reference implementations of PSkel for CPUs and GPUs were executed on the platforms described bellow. Compilation was done using GCC 5.4 (MPPA-256 and CPU) and NVCC version 8.0 (GPU) with the flags `-O3` (all platforms), `-march=native -mtune=native -ftree-vectorize` (CPU and GPU) and `-arch=sm_35` (GPU).

- **Xeon E5**: a desktop server featuring an Intel Xeon E5-2640 v4 (Broadwell) processor with 10 physical cores running at 2.4 GHz and 64 GB of RAM. Energy measurements on this platform are based on Intel's Running Average Power Limit (RAPL) interface, which considers the power consumption of hardware components through hardware counters. We used this approach to obtain the energy consumption of the CPU (`PACKAGE_ENERGY`) and DRAM (`DRAM_ENERGY`).
- **Tesla K40**: a NVIDIA Tesla K40c graphics board featuring 2880 CUDA parallel-processing cores with a base clock of 745 MHz and 12 GB of GDDR5 GPU memory. Energy measurements on this platform were obtained from NVIDIA Management Library (NVML). We used the NVML to gather the power usage for the GPU and its associated circuitry (*e.g.*, internal memory).

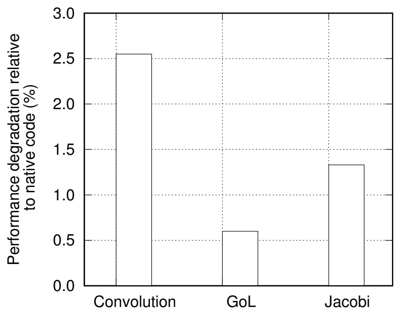


Fig. 5. Performance degradation of PSkel-MPPA (in percentage) relative to hand-optimized code for MPPA-256.

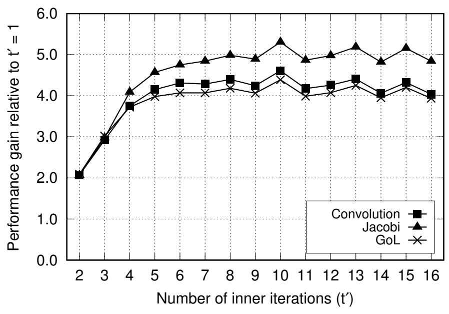


Fig. 6. Empirical study to find the best value for t'. The best trade-off is achieved with $t' = 10$.

We carried out several experiments with three stencil applications[1] implemented in PSkel: i) Convolution, which implements a classical convolution method used in signal and image processing; ii) GoL, which is a cellular automaton implementing Conway's Game of Life; and iii) Jacobi, which is an iterative method for solving matrix equations [4]. We also considered four input data sizes (2048x2048, 4096x4096, 8192x8192 and 12288x12288) to evaluate the performance of the aforementioned PSkel applications on MPPA-256, Xeon E5 and Tesla K40. Moreover, we evaluated the performance impacts of using different tile sizes (32x32, 64x64, 128x128 and 256x256) on MPPA-256, since input/output data sizes do not fit into compute clusters memory (2 MB). The maximum input/output and tile sizes were chosen carefully to fill MPPA-256 memories (2 GB of LPDDR3 and 2 MB of local memory in compute clusters). Finally, we fixed the number of iterations for each application to $t = 100$ in all experiments. All results represent averages of 20 runs with a maximum standard deviation of less than 1%.

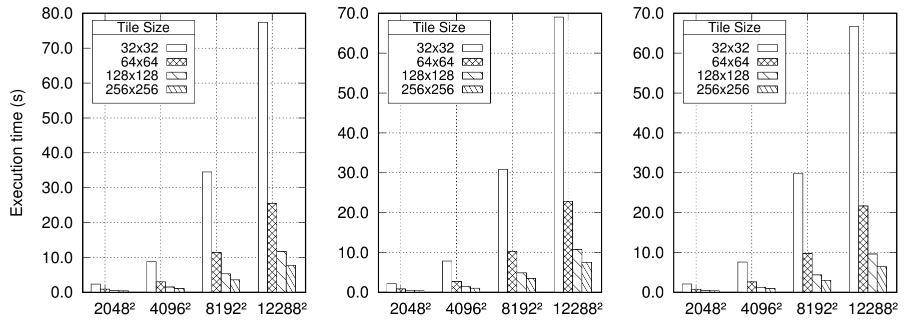
4.2 Overhead of PSkel

We first analyze the overhead introduced by our new back-end of PSkel (PSkel-MPPA). Fig. 5 shows the performance degradation of all three stencil applications implemented with PSkel-MPPA compared to hand-optimized ones implemented without PSkel abstractions. As it can be observed, the performance degradation introduced by PSkel-MPPA is minimal when compared to MPPA-256 native stencil code (less then 2.6%).

4.3 Sizing the Ghost Zone

In our solution, the trapezoidal tiling technique allows us to easily fine tune the size of the ghost zone with the t' parameter. Indeed, this is an important feature

[1] A detailed description about these PSkel applications is not presented in this paper due to space constraints but can be found in [11, 12].



(a) Convolution. (b) GoL. (c) Jacobi.

Fig. 7. Impact of tile size on the performance of the stencil applications.

to exploit the low amount of on-chip memory and to make a better use of the NoC available in low-power manycores. Thus, we carried out an empirical study with the aforementioned applications to determine the best value for this parameter. Fig. 6 shows the performance gains when varying t' from 2 to 16 (performance gains are relative to $t' = 1$). As we mentioned earlier, sizing the ghost zone poses a trade-off between the cost of redundant computations and the reduction in communication and synchronizations on the NoC. Our empirical study shows that the best trade-off is achieved when $t' = 10$ (performance obtained with $t' > 16$ varied around 4 and were omitted from the figure). Because of that, all results presented in next sections were carried out with $t' = 10$.

4.4 Tile Size vs. Performance

In this section, we analyze the impact of the tile size on the performance of PSkel applications on MPPA-256. Fig. 7 shows the performance of the stencil applications when varying the input data size and the tile size. Overall, we observed an average increase in the execution time of the applications between 2x and 3.3x as we double the input size. This behavior is expected since more communications and synchronizations must be performed for larger data inputs.

Moreover, we observed that the performance of the applications is greatly improved as we increase the tile size, regardless of the input size. The main reason for that is twofold. On the one hand, the number of `put`/`get` operations and synchronizations between the I/O and compute clusters on the NoC is greatly reduced as we increase the tile size. This allows for bigger data transfers per `put`/`get` operation, improving the NoC throughput. On the other hand, bigger tiles mean higher parallelism inside compute clusters (*i.e.*, OpenMP threads will have more work to compute), reducing the overhead imposed by OpenMP parallel regions. When varying the tile size from 32x32 to 64x64, we observed improvements of up to 3x on all applications. The performance gains increase to at least 6.9x and 10.3x when varying the tile size from 32x32 to 128x128 and from 32x32 to 256x256, respectively.

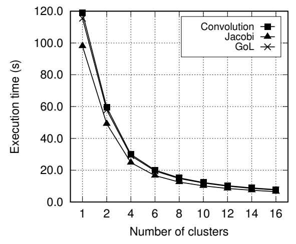


Fig. 8. Scalability.

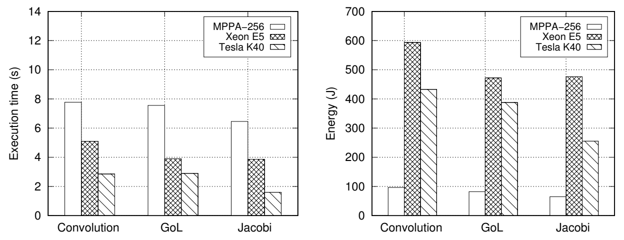


Fig. 9. Performance and energy comparison.

4.5 Scalability Analysis

Next, we analyze the scalability of PSkel-MPPA. Fig. 8 shows the execution time of each application on MPPA-256 when varying the number of compute clusters from 1 to 16. For this experiment, we used input data and tiles of size 12288x12288 and 256x256, respectively. As it can be noticed, all stencil applications present a similar behavior and have their execution times reduced as we increase the number of compute clusters. Overall, we observed a speedup gain of 15.3x with 16 compute clusters over the execution with a single compute cluster. This means that our solution is able to exploit all computing resources and the NoC of MPPA-256.

4.6 Comparison with CPU and GPU: Performance vs. Energy

Finally, we compare the execution time and energy consumption achieved by PSkel-MPPA against reference implementations of PSkel for CPU and GPU. In these experiments, we used input data of size 12288x12288. Based on the best performance achieved on Fig. 7, we used tiles of size 256x256 on MPPA-256. To make a fair comparison, we used the best tiling optimizations for Xeon E5 and Tesla K40 that were available in the multicore and GPU back-ends of PSkel, respectively. Fig. 9 presents the results obtained with all stencil applications.

Overall, PSkel-MPPA achieves competitive execution times compared to the CPU and GPU counterparts. As expected, the best performance was achieved on the GPU, since it has much more processing power than the other processors. The execution times of Convolution, GoL and Jacobi on MPPA-256 were 1.52x, 1.93x and 1.67x higher than on CPU, respectively. On the other hand, the execution times of Convolution, GoL and Jacobi on MPPA-256 were 2.72x, 2.61x and 4.04x higher than on GPU, respectively.

PSkel-MPPA achieved the best results with respect to the energy consumption on all applications. The main reason is that MPPA-256 offers a high parallelism and yet has a low power consumption. As we showed in Section 4.3, the trapezoidal tiling technique implemented in PSkel-MPPA was extremely important to achieve such energy improvements. We observed that the energy consumption on MPPA-256 was up to 7.34x and 4.71x lower than on the CPU and GPU, respectively.

5 Related Work

Due to the importance of parallel skeletons, and specifically the stencil parallel pattern, many recent efforts in research sought to improve the performance and broaden the support of skeletons on manycore processors. *Buono et al.* [1] ported a framework based on parallel skeletons, called FastFlow, to the manycore processor TilePro64. The TilePro64 has 64 identical processing cores interconnected by a mesh of network-on-chip. Similarly, *Thorarensen et al.* [16] presented a new back-end of the SkePU framework for the low-power manycore Myriad2. It features a heterogeneous architecture, targeting power constrained devices and mainly computer vision applications. *Lutz et al.* [9] used tiling techniques in stencil computations on multi-GPU environments by using the GPU memories collectively. Similarly, *Gysi et al.* [7] propose a framework for automatic tiling optimizations of stencil computations on CPU-GPU hybrid systems.

Recent works studied the performance and/or the energy efficiency of low-power manycore processors. *Totoni et al.* [17] compared the power and performance of Intel's Single-Chip Cloud Computer (SCC) to other types of CPUs and GPUs. Although they showed that there is no single solution that always achieves the best trade-off between power and performance, the results suggest that manycores are an opportunity for the future. *Morari et al.* [10] proposed an optimized implementation of radix sort for the Tilera TILEPro64 manycore processor. The results showed that their solution for TILEPro64 provides much better energy efficiency than an general-purpose multicore processor (Intel Xeon W5590) and comparable energy efficiency with respect to a GPU NVIDIA Tesla C2070. *Souza et al.* [14] proposed a benchmark suite to evaluate MPPA-256 manycore processor. The benchmark offers diverse applications regarding parallel patterns, job types, communication intensity and task load strategies, suitable for a broad understanding of performance and energy consumption of MPPA-256 and upcoming manycores. *Francesquini et al.* [5] evaluated three different classes of applications (CPU-bound, memory-bound and mixed) using highly-parallel platforms such as MPPA-256 and a 24-node, 192-core NUMA platform. They showed that manycore architectures can be very competitive, even if the application is irregular in nature. Using the Adapteva's Epiphany-IV low-power manycore, *Varghese et al.* [18] described how a stencil-based solution to the anisotropic heat equation using a two-dimensional grid was developed. This manycore has a low power budged (2 W) and has 64 processing cores. Similar to MPPA-256, Epiphany-IV has a very limited amount of local memory available to each core and no automatic prefetching mechanism exists; every data movement has to be explicitly controlled by the application.

To the best of our knowledge, PSkel-MPPA is the first complete implementation of a parallel stencil framework on MPPA-256. Our solution relieves programmers of the burden of explicitly dealing with NoC communications, the hybrid underlying programming model and the absence of cache coherence on MPPA-256. The trapezoidal tiling technique allows developers to fine tune the trade-off between the cost of redundant computations and the reduction in communication and synchronizations on the NoC when processing iterative stencil computations on MPPA-256.

6 Conclusion

Low-power manycores have emerged as a building block for constructing energy-efficient HPC platforms. However, the development of efficient parallel applications is very challenging on these processors because developers must deal with hybrid programming models, limited amount of directly addressable memory and NoC constraints. In this paper, we propose to ease the development of stencil applications on the low-power MPPA-256 manycore processor by means of parallel skeletons. More precisely, we proposed a new back-end of the PSkel stencil framework for MPPA-256 named PSkel-MPPA, providing a single high-level abstraction for stencil programming on CPUs, GPUs and MPPA-256. Our solution relieves programmers of the burden of explicitly dealing with communications and the hybrid underlying programming model of MPPA-256.

The trapezoidal tiling technique adopted in our solution was essencial to exploit the low-power MPPA-256 manycore processor, improving the performance of our solution. Our results showed that PSkel-MPPA achieved the best results with respect to the energy consumption on all applications, being up to 7.34x and 4.71x more energy efficient than on the CPU and GPU considered in this study, respectively. Moreover, PSkel-MPPA achieved competitive performance on MPPA-256 in comparison to the CPU and GPU reference implementations. The GPU achieved the best performance, since it has much more processing power than the other processors.

As future works, we intend to extend our support in PSkel-MPPA for 3D stencils. In this case, it would be necessary to consider a prefetching scheme to overlap communications with computations. Moreover, we intend to compare our results on MPPA-256 against low-power ARM processors, which may also include a low-power GPU. Finally, we intend to provide similar abstractions for dealing with other kinds of skeletons.

References

1. Buono, D., Danelutto, M., Lametti, S., Torquati, M.: Parallel patterns for general purpose many-core. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 131–139 (2013). https://doi.org/10.1109/PDP.2013.27
2. Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P.O., Méhaut, J.F.: Seismic wave propagation simulations on low-power and performance-centric manycores. Parallel Computing **54**, 108–120 (2016). https://doi.org/10.1016/j.parco.2016.01.011
3. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (Mar 2004)
4. Demmel, J.W.: Applied numerical linear algebra. SIAM (1997)

5. Francesquini, E., Castro, M., Penna, P.H., Dupros, F., de Freitas, H.C., Navaux, P.O.A., Méhaut, J.F.: On the energy efficiency and performance of irregular applications on multicore, NUMA and manycore platforms. J. Parallel Distrib. Comput. **76**, 32–48 (2014). https://doi.org/10.1016/j.jpdc.2014.11.002
6. Fu, H., et al.: The sunway taihulight supercomputer: System and applications. SCIENCE CHINA Information Sciences **59**(7), 1–16 (2016). https://doi.org/10.1007/s11432-016-5588-7
7. Gysi, T., Grosser, T., Hoefler, T.: MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In: International Conference on Supercomputing (ICS). pp. 177–186. ACM, Irvine, USA (2015)
8. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on GPU architectures. In: International Conference on Supercomputing (ICS). pp. 311–320. ACM, Venice, Italy (2012)
9. Lutz, T., Fensch, C., Cole, M.: PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. ACM Trans. Archit. Code Optim. **9**(4), 59:1–59:24 (2013)
10. Morari, A., Tumeo, A., Villa, O., Secchi, S., Valero, M.: Efficient sorting on the Tilera manycore architecture. In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 171–178. IEEE Computer Society, New York, USA (2012)
11. Pereira, A.D., Ramos, L., Góes, L.F.W.: PSkel: A stencil programming framework for cpu-gpu systems. Concurrency and Computation: Practice and Experience **27**(17), 4938–4953 (2015)
12. Pereira, A.D., Rocha, R.C.O., Castro, M., Goes, L.F.W., Dantas, M.A.R.: Extending OpenACC for efficient stencil code generation and execution by skeleton frameworks. In: International Conference on High Performance Computing & Simulation (HPCS). pp. 719–726. IEEE Computer Society, Genoa (2017). https://doi.org/10.1109/HPCS.2017.110
13. Rocha, R.C.O., Pereira, A.D., Ramos, L., Ges, L.F.W.: TOAST: Automatic tiling for iterative stencil computations on GPUs. Concurrency and Computation: Practice and Experience **29**(8), 1–13 (2017). https://doi.org/10.1002/cpe.4053
14. Souza, M.A., et al.: CAP bench: A benchmark suite for performance and energy evaluation of low-power many-core processors. Concurrency and Computation: Practice and Experience (2016). https://doi.org/10.1002/cpe.3892
15. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW). pp. 1176–1182. IEEE Computer Society, Shanghai, China (2011)
16. Thorarensen, S., Cuello, R., Kessler, C., Li, L., Barry, B.: Efficient execution of skepu skeleton programs on the low-power multicore processor myriad2. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). pp. 398–402 (2016). https://doi.org/10.1109/PDP.2016.123
17. Totoni, E., Behzad, B., Ghike, S., Torrellas, J.: Comparing the power and performance of intel's SCC to state-of-the-art CPUs and GPUs. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 78–87. IEEE Computer Society, New Brunswick, Canada (2012). https://doi.org/10.1109/ISPASS.2012.6189208
18. Varghese, A., Edwards, B., Mitra, G., Rendell, A.P.: Programming the Adapteva Epiphany 64-core network-on-chip coprocessor. In: International Parallel Distributed Processing Symposium Workshops (IPDPSW). pp. 984–992. IEEE Computer Society, Phoenix, USA (2014)