Portable exploitation of parallel and heterogeneous HPC architectures in neural simulation using SkePU

Sotirios Panagiotou
National Technical University of Athens, Greece
spanagiotou@microlab.ntua.gr

August Ernstsson
Linköping University, Sweden
august.ernstsson@liu.se

Johan Ahlqvist
Linköping University, Sweden
johan.ahlqvist@liu.se

Lazaros Papadopoulos
National Technical University of Athens, Greece
lpapadop@microlab.ntua.gr

Christoph Kessler
Linköping University, Sweden
christoph.kessler@liu.se

Dimitrios Soudris
National Technical University of Athens, Greece
dsoudris@microlab.ntua.gr

ABSTRACT

The complexity of modern HPC systems requires the use of new tools that support advanced programming models and offer portability and programmability of parallel and heterogeneous architectures. In this work we evaluate the use of SkePU framework in an HPC application from the neural computing domain. We demonstrate the successful deployment of the application based on SkePU using multiple back-ends (OpenMP, OpenCL and MPI) and present lessons-learned towards future extensions of the SkePU framework.

KEYWORDS

skeleton programming, neural simulation, HPC systems

1 INTRODUCTION

New generation HPC platforms consist of multiple heterogeneous cores. The end of Dennard scaling points towards more heterogeneity at hardware level, further increasing the complexity of applications. Examples include the upcoming exascale computing systems in USA and Europe, which will be massively parallel and highly heterogeneous, by integrating GPUs and various kinds of accelerators (Multipurpose Processing Array, FPGA, etc.). A recent trend is the combination of general-purpose CPUs with application-specific accelerators in large-scale systems. Various HPC application domains can benefit from this trend, including simulations and machine learning.

Programming parallel computing systems and accelerators is challenging. Effective exploitation of parallel systems requires taking into consideration synchronization, data locality and memory management issues, requiring significant programming effort by developers. With respect to acceleration, each type of accelerator has its own toolchain and programming model, forcing application developers to rewrite large parts of the codebase all over for each accelerator backend. Tools that assist application developers in the process of exposing parallelization and exploiting accelerators by reducing the required programming effort are highly desirable. SkePU[1] [10] falls in this category. It is an open-source skeleton programming framework for multicore CPUs and multi-GPU systems. (Algorithmic) *skeletons* [6] are generic parallelizable high-level programming constructs based on higher-order functions such as Map, Reduce, Stencil or Scan, which model common dependence and data access patterns and which can be parameterized in problem-specific sequential code. Skeletons provide a high degree of abstraction and portability with a quasi-sequential programming interface, as their implementations encapsulate all low-level and platform-specific details such as parallelization, synchronization, communication, memory management, accelerator usage and other optimizations. Each SkePU skeleton comes with a set of backends targeting the different supported platforms, including OpenMP (multicore CPU), OpenCL and CUDA (GPU) and StarPU-MPI (clusters).

Neural simulation, especially biophysically-detailed simulation, is a HPC application type presenting a very high computational load, due to both the numerical intensity of the models and the large space and time scale of the simulations. This immense computational load is evidently efficiently handled on HPC infrastructure; the required memory is provided by thousands of nodes, working in a single cluster [1], and computational performance has been greatly boosted by manycore [5], GPU [3] and reconfigurable [12] accelerators.

The insight that makes such massively-parallel implementations possible is that most parts of the neural network can be simulated in parallel within a single simulation step. To that end, there is a rich variety of data-parallel numerical schemes for time-driven simulation that have been devised since the dawn of supercomputing, and that also apply to neural simulations.

Computer simulation is a crucial part of the neuroscience research process. Simulation speed is also important on another aspect of neuroscience: It constrains the complexity of neuron models that can be explored. Neural models can only be as large and as wide in time-scale as they still can be simulated within a practical timeframe. So, computational performance defines the scale of feasible neural simulations; more so that even available computer memory does, due to the high numerical intensity of the calculations.

[1] SkePU: https://www.ida.liu.se/labs/pelab/skepu

This work is about the portable exploitation of HPC parallel computing systems and accelerators by a neural simulator using the SkePU programming framework. By using a state-of-the-art high-level parallel programming framework, such as SkePU, the intra-node parallelism of the application is exposed and the application is deployed on a multicore CPU system. Additionally, the single-node GPU implementation of the application is evaluated. Finally, by using SkePU, the application is deployed on multiple nodes as well. Apart from presenting the evaluation results for each backend, this work also focuses on describing our experiences and lessons learned by the use of SkePU in the neural simulator.

The rest of the paper is organized as follows: Section 2 introduces the SkePU programming framework, and gives a short description of the neural simulator. The experimental results are presented in Section 3, while in Section 4 we describe our experiences towards possible directions for future extensions of SkePU. Finally, in Section 5 we draw our conclusions.

2 SKELETON PROGRAMMING, SKEPU AND BRAIN MODELING APPLICATION

The skeleton programming framework *SkePU* [10], being developed at Linköping University, provides a single, sequential-like programming interface extending modern C++ with data-parallel skeletons. For each skeleton, implementation variants (backends) for sequential C++, OpenMP, CUDA and OpenCL are available. Currently, an additional backend for clusters targeting the MPI bindings in the task-based *StarPU* runtime system [2] is being developed; first results for it are reported in Section 3. In the most recent version of SkePU, the customization mechanism of skeletons by problem-specific user code has recently been generalized to include externally defined platform-specific custom user functions, which allow platform-experts to leverage platform-specific SIMD instructions or special instructions without breaking the universal portability of the SkePU source code [9]. An independent recent study confirmed the programmability improvements of SkePU compared to thread-based programming frameworks at insignificant performance overheads for dataparallel PARSEC benchmarks [8].

Smart data-containers [7] are STL-like C++ data structures such as `Vector` and `Matrix` that wrap array-based operands passed to and from skeleton instance calls in SkePU code. They allow SkePU to transparently perform runtime optimizations of data movement to/from device memories, device memory allocation, and data locality optimizations across dependent skeleton calls.

The Brain modeling application is a time-driven simulator of biophysically detailed, Extended Hodgkin-Huxley (eHH) [11] models of individual neurons. It is a ground-up new design that focuses on supporting large-scale networks on HPC infrastructure. It is currently under development, and aims to support the entire NeuroML standard [4]. The main features of this type of models are:

- The state variables and parameters involved in the models correspond directly to electrochemical quantities, such as membrane potential, concentration of various ions, neuromodulators etc. across the neuron.
- The chemical state of a neuron varies across its extent, allowing investigation of how action potentials are propagated through the dendrites and axon, and how different parts of a cell have different properties.

Thanks to these properties, the Extended Hodgkin-Huxley neural models are an effective tool to study the chemistry behind neural activity, to investigate how the electrochemical dynamics facilitate neural computation and to examine the behaviour of neurons changes under off-nominal physiological factors, such as drug effects or medical conditions.

3 PARALLELIZATION AND ACCELERATION OF BRAIN MODELING USING SKEPU

3.1 SkePU applied in Brain modeling

We used SkePU in the brain modeling miniapp, to target multiple parallelization platforms (single-node OpenMP and OpenCL GPU, and multi-node MPI+OpenMP). The implementations generated by SkePU for each target were then evaluated it terms of computational performance. The original application code, and a basic OpenMP-enabled version of that code were used as performance baselines.

As a first step for SkePU integration and evaluation, a SkePU mini-app for the Brain Modeling workload was developed. The mini-app is the most computationally intensive kernel of the original application and simulates a tightly-connected cluster of similar neurons, which interact with each other through continuous-time graded synapses. Each neuron is lumped into a single finite element. The electro-chemical dynamics follow a fixed set of Hodgkin-Huxley equations for each neuron, which is a common reference point for biological neural network simulators. The weight of synapses between neurons is stored on a square matrix, and simulation of each neuron requires the respective row of a matrix to estimate the influence of adjacent neurons.

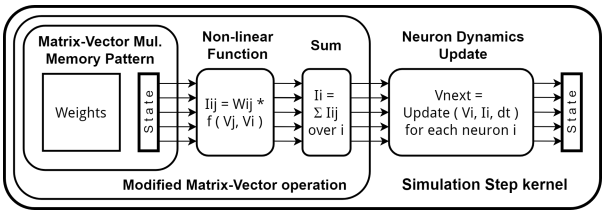

Figure 1: Brain Modeling dataflow

The parallelism inherent in the simulation kernel, and the relevant memory access patterns of the workflow, are demonstrated in a comprehensive dataflow diagram in Figure 1. Note that V represents the neuron's state, and I represents synaptic current between neurons.

In the span of a simulation step, each individual neuron passes through two main stages of processing:

- The first stage is, for a given neuron, to calculate the total synaptic electrical current, flowing from other neurons connected to this neuron.
- The second stage is to use the total synaptic current, along with the neuron's internal electrochemical state and injected stimulus, to advance the neuron's state to the next timestep, according the neuron's differential equations.

The second part is straightforward to parallelize; computations for each neuron run independently, and work on separate parts of data. However, the first part is more challenging to accelerate: to calculate synaptic current, computation for each neuron needs to have random access to the state of other neurons. This adds a large volume of memory traffic to the processing, and asserts the need for each simulated neuron to have a synchronized view of other neurons' states, for each timestep.

The current flow between neurons is modeled by a non-linear formula for this mini-app. This current is weighted by the corresponding value in weight matrix, and summed for each neuron.

Thus, the synaptic current part of the simulation kernel has a memory access pattern very similar to the Matrix-Vector Multiplication linear algebra operation. The differences are that two different elements of the neuron's state are used in the inner products with the matrix weights, and that each inner product is passed through a non-linear function before summation. These differences preclude using the common optimized linear-algebra libraries; which underscores the usefulness of a code-transformation middleware like SkePU, which can automatically synthesize high-performance code for the case in point.

3.2 Evaluation

The main part of the simulation is a loop: repetitive estimation of the state of all neurons in the next timestep, until the whole duration of the simulation is simulated. Estimating each neuron's state depends only on the present state of all neurons. Thus, simulation of each neuron can be performed independently within a single timestep.

In our SkePU-enabled implementation of the miniapp, we leveraged the acceleration capabilities of the SkePU framework by expressing this parallelism of computations per individual neuron. Simulation of neurons in a timestep is done as a parallel task, with each work item consisting of the current status of a neuron, and the code to compute the neuron's state after the timestep.

Since each neuron is simulated individually, the Map skeleton was employed. The simulation kernel essentially maps each neuron's state in the current timestep, to its updated state for the next timestep. Since neurons communicate with each other through the connectivity (weight) matrix, both the connectivity matrix and the neuron state vector were passed as auxiliary read-only arguments, to be used when simulating each neuron.

Another way to conceptualize the Map skeleton is to consider how the original pseudocode was transformed, in Figure 2: the outer loop was converted to a Map kernel invocation, and the code inside that loop was transferred to the kernel's inner `get_stateNext` function. In the process, the fact that the loop body produces a single neuron's updated state is made explicit, since this is the only value the `get_stateNext` function produces.

After applying the Map skeleton, we evaluated the SKEPU-ized miniapp under the OpenMP and OpenCL backends, on a single node. The node integrates twin Intel Xeon Gold 6138 processors (40 cores, 80 hyperthreads total), a nVidia Tesla V100 GPU(with 32GB on-board memory) and 128 GB of main system memory. Run times for various neural network sizes are shown in Figure 3.

The simulations ran for 200 simulation steps of 10 μsec simulated time. The maximum network size examined was 90000 neurons,

Original code:

```
for each timestep:
  for each neuron i:
    Get Isyn from adj. Neurons
    … internal dynamics code …
    Compute stateNext[i]
```

SkePUized code:

```
Function get_stateNext(i, stateNow, adj.matrix):
  Get Isyn from adj. Neurons
  … internal dynamics code …
  Return stateNext for neuron I

Kernel = Map(get_stateNext)
for each timestep:
  stateNext = Kernel(stateNow, adj.matrix)
```

Figure 2: Using SkePU in brain modeling (pseudocode)

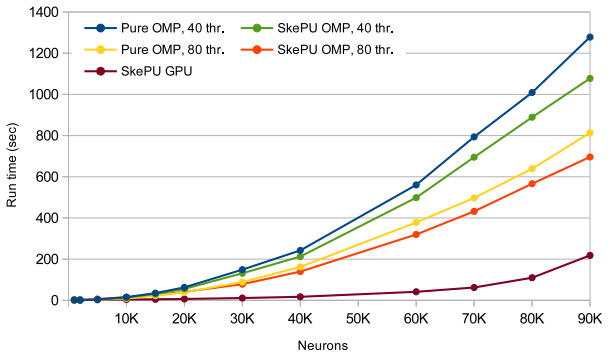

Figure 3: Execution time using the OpenMP and OpenCL backends

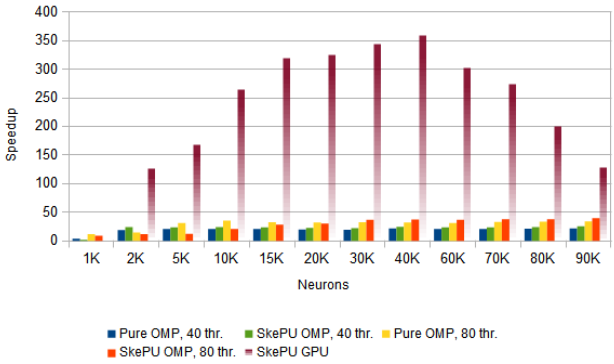

Figure 4: Speedup vs. single-threaded miniapp

which fully utilizes the GPU's memory for all-to-all connectivity. CPU-based OpenMP acceleration was evaluated under 40 and 80 threads, to assess the effect of hyperthreading on performance.

We observe that the SkePU's GPU backend outperforms the rest, reaching a 10x speedup over the best CPU-based implementation, for up to 40000 neurons. For larger network sizes, GPU performance starts to degrade down to 3x the speed of the best CPU implementation. Thus, it is likely some part of the architecture becomes a system bottleneck for the GPU, and profiling of such cases will help determine the issue and to incorporate the resulting solution into SkePU, for general use.

Another observation is that the SkePU-ized OpenMP backend performs better than the OpenMP backend, when 40 threads are run and when the network is large. This probably is because the

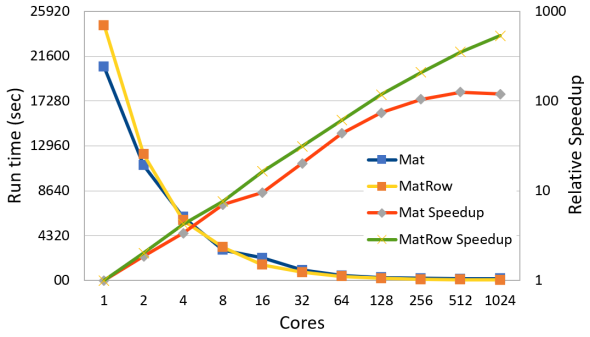

Figure 5: Mini-app using the MPI backend on up to 32 nodes.

SkePU backend uses more aggressive defaults in OpenMP tuning parameters; the baseline implementation mostly used OpenMP defaults and did not try optimizing for the miniapp's case.

Figure 5 shows the scaling behavior of the mini-app for the StarPU-MPI backend on the CPU cluster *Tetralith*, using up to 32 nodes each with 32 cores (in 2 Intel Xeon Gold 6130 CPUs) and 96GiB main memory, interconnected by a 100 GiB/s Intel Omni-Path network (fat-tree), for 90K neurons, 200 time steps and density 0.12. Here the user function of the `Map` skeleton uses the new SkePU-3 `MatRow` proxy of the `Matrix` data-container for 1D-distributable matrix operands, which avoids the expensive broadcast communication pattern that results from using the general `Mat` container proxy. We observe that the very good intra-node scaling (using OpenMP-only up to 32 cores) flattens out with `Mat` across multiple nodes but continues with `MatRow` at only slightly lower slope, and a speedup of 531x is achieved with 1024 cores (vs. 119x with `Mat`).

4 DISCUSSION AND LESSONS LEARNED

Portability is an increasingly important quality for HPC application developers. In systems with various accelerators, having the opportunity to evaluate applications in each of them without code modifications is very important. The SkePU portability feature enables developers to maintain a single application codebase, which is portable to various systems and platforms.

SkePU fits very well in applications which are dominated by *data parallelism*. The Brain Modelling workload is highly data-parallel, however it presents the following challenges with respect to using SkePU framework:

The heterogeneity of the equations involved among different parts of different neurons means that different work items may use different code kernels to run. At the same time, the *formulas in these equations may change frequently* while novel models for neurons are being developed and evaluated. So for effective use by neuroscientists, the toolchain and code-generation considerations of the SkePU package should be encapsulated in a user-friendly package.

The *sparse connectivity* between parts of each cell is a common feature in neural simulation. The random-access considerations and impact on each hardware backend must be encapsulated and taken care of by the SkePU package, since the acceleration strategies crucially depend on the underlying hardware architecture. To that end, SkePU is being extended with a `SparseMatrix` data container with row-compressed storage format, which can then be employed instead of the dense `Matrix` container.

The work items to be run in parallel often have *irregular corresponding data sizes*. While an upper limit on such sizes can be enforced and the extra data size be replaced with dummy components, this may not always be the appropriate case for the hardware backends and it introduces SkePU-specific changes into the original application.

Finally, using the SkePU-3 `MatRow` container for distributable matrix operands was found significant for scaling up to multi-node or multi-GPU configurations.

5 CONCLUSION

We evaluated the use of an advanced programming framework, SkePU, in a neural computing application. We demonstrated the flexibility that SkePU provides: the execution of an application in different HPC architectures using a single application codebase. Programming frameworks such as SkePU that express application parallelism and enable portability across a wide range of different HPC platforms are very useful for application developers, considering the complexity of modern HPC applications and architectures.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 801015 (EXA2PRO, www.exa2pro.eu). We thank NSC Linköping and SNIC for access to the Tetralith cluster (SNIC 2016/5-6).

REFERENCES

[1] R. Ananthanarayanan, S. K. Esser, H.D. Simon, and D.S. Modha. 2009. The Cat is out of the Bag: Cortical Simulations with 10^9 Neurons, 10^{13} Synapses. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*.

[2] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency Comput.: Pract. Exp.* 23 (2011), 187–198. Issue 2.

[3] R. Ben-Shalom, N.S. Athreya, C. Cross, H. Sanghevi, A. Korngreen, and K.J. Bender. 2019. NeuroGPU, software for NEURON modeling in GPU-based hardware. *bioRxiv* (2019).

[4] R.C. Cannon, P. Gleeson, S. Crook, G. Ganapathy, B. Marin, E. Piasini, and R.A. Silver. 2014. LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Frontiers in Neuroinformatics* 8 (2014), 79.

[5] G. Chatzikonstantis, H. Sidiropoulos, C. Strydis, M. Negrello, G. Smaragdos, C.I. De Zeeuw, and D.J. Soudris. 2019. Multinode implementation of an extended Hodgkin–Huxley simulator. *Neurocomputing* 329 (2019), 370 – 383.

[6] M. Cole. 2004. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.* 30, 3 (2004), 389–406.

[7] U. Dastgeer and C. Kessler. 2016. Smart containers and skeleton programming for GPU-based systems. *Int. J. of Parallel Progr.* 44 (June 2016), 506–530. Issue 3.

[8] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto. 2017. Bringing Parallel Patterns Out of the Corner: The P3ARSEC Benchmark Suite. *ACM Trans. Archit. Code Optim.* 14, 4, Article 33 (2017), 33:1–33:26 pages.

[9] A. Ernstsson and C. Kessler. 2020. Multi-variant User Functions for Platform-aware Skeleton Programming. In *Parallel Computing: Technology Trends, series: Advances in Parallel Computing, vol. 36*. IOS press, 475–484. Proc. of ParCo-2019 conference, Prague, Sep. 2019.

[10] A. Ernstsson, L. Li, and C. Kessler. 2018. SkePU-2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming* 46, 1 (01 Feb 2018), 62–80.

[11] E.R. Lewis. 1966. Neuroelectric potentials derived from an extended version of the Hodgkin-Huxley model. *Journal of Theoretical Biology* 10, 1 (1966), 125 – 158.

[12] A. Sripad, G. Sanchez, M. Zapata, V. Pirrone, T. Dorta, S. Cambria, A. Marti, K. Krishnamourthy, and J. Madrenas. 2018. SNAVA—A real-time multi-FPGA multi-model spiking neural network simulation architecture. *Neural Networks* 97 (2018), 28 – 45.