Lapedo: Hybrid Skeletons for Programming Heterogeneous Multicore Machines in Erlang

Vladimir JANJIC, Christopher BROWN, Kevin HAMMOND
School of Computer Science, University of St Andrews, UK.
{vj32, cmb21, kh8}@st-andrews.ac.uk

Abstract. We describe *Lapedo*, a novel library of *hybrid* parallel skeletons for programming *heterogeneous* multi-core/many-core CPU/GPU systems in Erlang. Lapedo's skeletons comprise a mixture of CPU and GPU components, allowing skeletons to be flexibly and dynamically mapped to available resources, with all of the low-level tedious code to divide work between CPUs and GPUs, transfer the data between the main and GPU memory and offload computations to the GPUs provided by the library. We evaluate the effectiveness of *Lapedo* on three realistic use cases from different domains, demonstrating significant improvements in speedups compared to CPU-only and GPU-only executions.

Keywords. Parallel skeletons, Hybrid skeletons, Heterogeneous multicore systems, GPU offloading

1. Introduction

Following the initial stages of the multi-core revolution, further major changes in computer hardware are now ongoing. Hardware is getting increasingly heterogeneous, integrating *accelerators*, such as graphic processing units (GPUs), field programmable gate arrays (FPGAs) and even lightweight many-core CPU accelerators, with traditional multi-core processors. These *heterogeneous* systems have a potential to deliver orders of magnitude more performance than traditional CPU-only based systems, and are increasingly found in high-performance architectures. In order to fully exploit the potential that these systems offer, programmers needs to combine several different low-level programming models, e.g. OpenCL for GPUs, VHDL or Verilog for FPGAs and OpenMP for CPUs. They must also explicitly manage data transfers between main memory and accelerator memory, schedule computations, fetch results, etc. Moreover, the solutions that perform optimally are usually tied to a specific heterogeneous architecture and cannot easily be ported, yielding problems in terms of e.g. increased maintenance costs and lack of longevity. This makes programming heterogeneous multicore/many-core systems *extremely difficult and complex* compared with programming multi-core CPUs. What is needed are high-level programming abstractions, hiding the hardware complexity of such systems by abstracting over the varying low-level models, while still achieving (near-)optimal accelerated performance.

This paper presents *Lapedo*[1], a novel system of parallel *skeletons* for programming heterogeneous multi-core/many-core systems in the functional language Erlang. Functional programming approaches naturally provide high-level abstractions through e.g. higher-order functions. In *Lapedo*, we exploit this to build skeletons: parameterised parallelism templates that abstract over low-level parallelism details and parallel structures of a program. Hybrid skeletons contain alternative implementations of their components for different processor types, automatically providing tedious and error-prone code for transferring data to/from processor types, scheduling, fetching results, etc. *Lapedo* also provides mechanisms for dividing work between different processor types, ensuring load balancing and eliminating the need for extensive profiling and performance tuning. This allows skeletons to be flexibly deployed on an arbitrary combination of CPUs and GPUs, allowing us to achieve performance results that are better than either CPU or GPU execution alone, while still keeping a very high-level of abstraction. Although in this paper we focus only on GPU accelerators, our low-level implementation is based on OpenCL and our work can, therefore, be easily extended to a wide range of other accelerators, including Intel Xeon PHIs, FPGAs and DSPs. The general hybrid skeleton approach can also be applied to other language frameworks, such as C++, Java, Haskell, etc. This paper makes the following research contributions:

1. we describe the *Lapedo* Erlang library of *hybrid skeletons* that allow CPU and accelerator components to be combined within the same skeleton;
2. we describe the current *Lapedo* implementation on heterogeneous CPU/GPU combinations; and,
3. we demonstrate that *Lapedo* allows us to produce efficient and scalable code for heterogeneous systems, achieving real speedups of up to 21.2 over sequential Erlang programs on a 24-core machine with a GPU.

2. Heterogeneous Parallelism and Skeletons

Compared to traditional CPUs, accelerators usually offer higher-performance (in terms of e.g. FLOPS) at lower clock speeds and with reduced energy usage per unit of performance. However, they are usually restricted in terms of the parallelism model that is offered (often only data-parallel) and can be much more difficult to program than traditional CPU-only systems. This creates a significant barrier for applications programmers. In this paper, we will restrict our attention to CPU/GPU combinations, representing the current most widely-used class of heterogeneous multicores. However, since our implementations target the OpenCL programming model, which is supported by other types of accelerators (such as FPGAs), our work is also applicable in wider settings.

[1] Named after the hybrid *Lapedo* or *Lagar Velho* Neanderthal/Homo Sapiens skeleton.

Conventional Approaches to GPU Programming The two most widely-used approaches to GPU programming, CUDA and OpenCL provide similar portable, but low-level SIMD programming interfaces. Unfortunately, *programmability* is still generally lacking with these models: the programmer needs to take care of a number of very low-level programming aspects, including the number of threads and thread blocks, data transfers between CPUs and GPUs and scheduling of computations on the GPUs. Some newer standards, such as SyCL[2], aim to further simplify GPU programming by offering further abstractions. In addition, there are several algorithmic skeleton libraries (see Section 2.2) for programming GPUs, such as SkePU [7] and SkeCL [15]. However, these models are either restricted to GPUs only or require the programmer to have a deep understanding not only of the problem that is being solved, but also of the underlying hardware architecture. This usually results in a solution that is heavily optimised for a particular hardware system and which, therefore, lacks *performance portability.* The GPU-specific code is also often highly fragile and likely to be error-prone.

Hybrid Programming Approaches There are several models that support combining CPUs and GPUs for the same computation, including mapping of work to different devices, and thus allowing proper "hybrid" programming. StarPU [5] is a runtime system that supports CPU/GPU systems and is used as one of the backends for SkePU. It maps the SkePU skeleton calls to one or more *tasks* that can be mapped to different devices. It was shown to give good performance for simple benchmarks on CPU/GPU configurations [10]. Compared to this approach, the library that we present in this paper is higher level, thanks to the use of functional programming technology (see below). Grewe and O'Boyle proposed a purely static partitioning scheme for OpenCL [9], based on predictive modelling and program features. While this enables deriving good mappings, the programming model is still OpenCL, making it too low-level for non-experts. Qilin [11] is a heterogeneous programming system that includes an automatic adaptive mapping technique to map computations processing elements on CPU/GPU machines. We use a very similar mechanism for deriving a division of work between CPUs and GPUs in Section 3.1.

2.1. (Heterogeneous) Parallel Programming in Erlang

Erlang [1] is a strict, impure, functional programming language. It is widely used in the telecommunications industry, but is also beginning to be used more widely for high-reliability/highly-concurrent systems, e.g. databases [12], AOL's *Marketplace by Adtech* [16], and WhatsApp [13]. It has excellent support for concurrency and distribution, including built-in fault-tolerance. Erlang supports a threading model, where *processes* model small units of computation. The scheduling of processes is handled automatically by the Erlang Virtual Machine, providing basic load balancing mechanisms. We build on these lower-level mechanisms to provide higher-level parallelism abstractions, using *algorithmic skeletons.* We exploit all the usual Erlang distribution mechanisms to build highly-distributed scalable systems, where individual nodes can exploit accelerators using *Lapedo.*

[2] `https://www.khronos.org/sycl`

Accelerator Programming in Erlang Erlang has no native support for programming accelerators. However, a library containing OpenCL bindings is available [14], which provides an Erlang interface to low-level OpenCL functions to set up accelerator computations, transfer data to/from the accelerators, and launch kernels implemented in OpenCL, plus basic marshalling mechanisms between *binary* data structures in Erlang and C arrays. While enabling programmers to write their code in Erlang, this library does not simplify GPU programming, since the programmer is still required to write code that is equivalent to programming directly in OpenCL. In the *Lapedo* library, we build on this library to provide higher-level skeletons that encapsulate most of the required OpenCL code.

2.2. Skeletons

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction [4] into parameterised templates. For example, we might define a *parallel map* skeleton, whose functionality is identical to a standard *map* function, but which creates a number of Erlang processes (*worker processes*) to execute each element of the map in parallel. Using a skeleton approach allows the programmer to adopt a top-down *structured* approach to parallel programming, where skeletons are composed to give the overall parallel structure of the program. Details such as communication, task creation, task or data migration, scheduling, etc. are embedded within the skeleton implementation, which may be tailored to a specific architecture or class of architectures. This offers an improved level of portability over typical low-level approaches. A recent survey of algorithmic skeleton approaches can be found in [8].

2.3. The Skel *Library for Erlang*

Lapedo is integrated into the *Skel* [3,2] library, which defines a small set of classical skeletons for Erlang. Each skeleton operates over a stream of input values, producing a corresponding stream of results. Skel also allows simple composition and nesting of skeletons. We consider the following skeletons:

- `func` is a simple wrapper skeleton that encapsulates a sequential function as a streaming skeleton. For example, in Skel `{func, fun f/1}` denotes a `func` skeleton wrapping the Erlang function, `f`, with `f/1` denoting the arity of `f`. In this paper, we denote `func` skeleton simply by `{func, fun f}`. When *func* is instantiated, a new Erlang process is created and mapped to a single OS thread, which executes the function sequentially without any parallelism.
- `pipe` models a composition of skeletons $s_1, s_2, \ldots, s_n$ over a stream of inputs. Within the pipeline, each of the s_i is applied in parallel to the result of the s_{i-1}. For example, in Skel: `{pipe, [{func, fun f}, {func, fun g}, {func, fun h}]}` denotes a parallel pipeline with three stages. Each pipeline stage is a `func` skeleton, wrapping the Erlang functions, `f`, `g`, and `h`.
- `farm` models application of the same operation over a stream of inputs. Each of the n farm *workers* is a skeleton that operates in parallel over independent values of the input stream. For example, in Skel: `{farm, 10, {pipe, [{func, fun f}, {func, fun g}]}}` denotes a `farm` where

the worker is a parallel pipeline with two stages (each the `func` skeleton wrapper for the functions `f` and `g`, respectively). This example `farm` has 10 workers, as specified by the second parameter, therefore running 10 independent copies of the pipeline skeleton.

- `cluster` is a data parallel skeleton, where each independent input, x_i can be partitioned into a number of sub parts, $x_1, x_2, \ldots, x_n$, that can be worked upon in parallel. A skeleton, s, is then applied to each element of the sub-stream in parallel. Finally the result is combined into a single result for each input. An example, in Skel: `{cluster, {func, fun f}, fun dec, fun rec}` denotes a `cluster` skeleton with each worker a simple sequential function, where the `dec` function is used to decompose each input list into chunks, and the `rec` function is used to recompose the list of result from the chunks of results. This is similar to the `farm` skeleton example above, except the number of workers is not specified for the `map` skeleton but rather implicitly computed by the `dec` function. Note that `decom` and `recom` can be identity functions, in which case we get the usual `map` skeleton applied to each element of input stream, where the `func` skeleton is applied to each element of an input list in parallel.
- `feedback` wraps a skeleton s, feeding the results of applying s back as new inputs to s, provided they match a filter function, f.

3. The Lapedo System for Hybrid Skeletons

Lapedo extends the Skel library described in Section 2.3 with hybrid versions of the *farm* and *cluster* skeletons that combine CPU and GPU *components*, which are expressed by wrapping operations using the `func` skeleton. This ensures that operations written for a CPU will be mapped to single sequential OS thread, and GPU operations will be mapped to a GPU device. In general, a programmer is required to provide these components which, in the case of the GPU, contain all the code for creating buffers, transferring data to and from the GPU and scheduling the kernel that implements the actual operation. *Lapedo* provides a mechanism for automatically generating this boilerplate code, and, in order to use the hybrid skeletons, a programmer is only required only to write Erlang CPU components and relatively simple problem-specific GPU kernels in OpenCL.

Hybrid Farm. Similarly to the CPU-only farm skeleton (Section 2.2), hybrid farm applies the same operation to a stream of inputs in parallel. It requires two skeleton instances that provide implementations of the operation for a sequential CPU thread and a GPU. Each element of the input stream is tagged with either `cpu` or `gpu` tag[3] and, depending on this tag, sent to one of the two inner skeletons. In this way, different processor types process input elements in parallel. The syntax of the hybrid farm skeleton is

{hyb_farm, CPUSkeleton, GPUSkeleton, NCPUWorkers, NGPUWorkers}

[3] In the future, additional tags will be supported to accommodate additional accelerator types.

where `NCPUWorkers` and `NGPUWorkers` are the number of instances of the `CPUSkeleton` and `GPUSkeleton` that are created. These determine how many input elements will be tagged with the `cpu` and `gpu` tags. For example, if there are 20 input tasks, and `NCPUWorkers` is 4 and `NGPUWorkers` is 1, then 16 tasks will be tagged with the `cpu` tag and 4 will be tagged with the `gpu` tag. For example

```
{hyb_farm, {func, fun f_CPU/1}, {func, fun f_GPU/1}, 4, 1}
```

defines a hybrid farm skeleton, where the operation is a simple function, `f_CPU` being a sequential CPU operation and `f_GPU` for a GPU operation. As mentioned above, the code for `f_GPU` can be generated automatically, based on a programmer-provided OpenCL kernel that implements a function equivalent to `f_CPU`.

Hybrid Cluster. Similarly to the Section 2.2, we focus on a list version of the hybrid cluster skeleton, where each element in an input stream is a list, and each of these lists is decomposed into sublists that are processed in parallel. *Lapedo* also provides a more general version of this skeleton, that works on arbitrary data structures. The syntax of the hybrid cluster skeleton is

```
{hyb_cluster, CPUSk, GPUSk, DecompFun, RecompFun, NCPUW, NGPUW}
```

As in the case of the hybrid farm, `CPUSk` and `GPUSk` provide CPU and GPU implementations of the operation that is to be applied to the sublists (generated by `DecompFun`) of each list of an input stream. Decomposing an input list appropriately in the case of the hybrid cluster is usually non-trivial, due to a difference in performance of a CPU thread over a GPU for a given problem. For this reason, we provide two variants of the hybrid cluster skeleton that automatically find a good decomposition of work:

- `{hyb_cluster, CPUSk, GPUSk, ChunkSizeMult, TimeRatio, NCPUW, NGPUW}`, where `ChunkSizeMult` is a minimal length of each sublist, with the length of each sublist after decomposition being its multiplier. `TimeRatio` is a ratio between the processing time of a single sublist of size `ChunkSizeMult` on a CPU and on a GPU (which can be obtained using profiling). This parameter determines how much faster the GPU is in processing work than a CPU thread (or vice versa). `NCPUW` and `NGPUW` determine how many sublists will be processed by sequential CPU threads and how many by GPUs. These two parameters also determine the total number of chunks that each task is decomposed into (`NCPUW`+`NGPUW`) and, together with `TimeRatio`, determine the length of each sublist. For more details about how lengths of sublists are calculated, see Section 3.1
- `{hyb_cluster, CPUSk, GPUSk, ProfChunk, NCPUW, NGPUW}`, which is similar to the above version, with the difference that `ChunkSizeMult` and `TimeRatio` parameters are here automatically calculated by doing profiling on a user provided example sublist `ProfChunk`, which needs to be representative of the sublists that will be processed by `CPUSk` and `GPUSk`.

3.1. Division of Work Between CPUs and GPUs.

The `hyb_cluster` skeleton requires the numbers of CPU and GPU workers to be specified explicitly (`NCPUW` and `NGPUW` parameters). Where there is no nesting of skeletons, i.e. where there is only a `hyb_cluster` skeleton at the top level, and the `CPUSk` and `GPUSk` skeletons are simple `func` skeletons, we can simply set `NCPUW` and `NGPUW` to be the number of CPU cores and GPU devices in the system, respectively. The problem with this, however, is that for suitable problems, GPUs are much faster in processing tasks than CPU cores. Therefore, if we divide input lists into equally-sized sublists, the same amount of work will be assigned to each CPU and GPU worker, in which case GPUs will finish much faster than CPU cores, resulting in load imbalance.

The aforementioned problem can be avoided if we do a smarter decomposition of input lists. Assuming that a given problem is regular (i.e. that it takes the same amount of time to process each element of an input list) and that we can obtain timing information (e.g. using profiling) to determine how much faster can a GPU process a set of list items than a CPU core (including time to transfer the data to/from the GPU memory), we can, using some simple formulae, derive how many list items should be processed by the GPUs and how much by each CPU core in order to get the best execution time. For example, assume that we have g GPUs and c CPU cores in a system, and that the ratio between processing time for k items between a CPU and a GPU is ratio. If an input list has n items (where n is divisible by k), then we can estimate the time it takes to process all of the items in the list if n_c items are processed by CPU cores by

$$T(n_c) = \max\left\{ \left\lceil \frac{\frac{n_c}{k} \cdot \text{ratio}}{c} \right\rceil, \left\lceil \frac{\frac{n-n_c}{k}}{g} \right\rceil \right\},$$

where the first argument of the max is the time it takes to process n_c items on CPU cores, and the second argument is the time it takes to process the remaining items by the GPUs. The best time we can obtain is then $\min\{T(n_c) | n_c \in \{0, k, 2k, ..., n\}\}$, and the optimal number of items to process on CPU cores is such n_c for which this minimum is obtained. In this way, we calculate a pair $(n_c, n - n_c)$ for the number of list items to be processed by CPU cores and the GPUs, respectively. Sublists lengths sizes for CPU cores are then

$$\left\{ \underbrace{\left\lfloor \frac{n_c}{c} \right\rfloor, \left\lfloor \frac{n_c}{c} \right\rfloor, \dots \left\lfloor \frac{n_c}{c} \right\rfloor}_{c-(n_c \bmod c) \text{ times}}, \underbrace{\left\lceil \frac{n_c}{c} \right\rceil, \left\lceil \frac{n_c}{c} \right\rceil, \dots, \left\lceil \frac{n_c}{c} \right\rceil}_{n_c \bmod c \text{ times}} \right\}.$$

We can similarly calculate the chunk sizes for the GPUs. The parameter k above should be chosen so that it gives the best parallelism on the GPU, i.e. it should be maximum number of list items that the GPU can process in parallel (parameter `ChunkSizeMult` in the description of `hyb_cluster` skeleton).

4. Evaluation

We evaluate *Lapedo* on three realistic use cases: *Ant Colony Optimisation*, *Football Simulation* and *Image Merging*. The experiments were conducted on a system that comprises two 12-core 2.3GHz AMD Opteron 6176 processors, 32GB RAM and NVidia Tesla C2050 GPU with 448 CUDA cores. We evaluate the speedups relative to the sequential Erlang versions. To eliminate the effect of variation of the execution times for CPU components on the same configuration (i.e. with the same number of cores used), each data point is averaged over 10 application runs with the same input data.

4.1. Ant Colony Optimisation

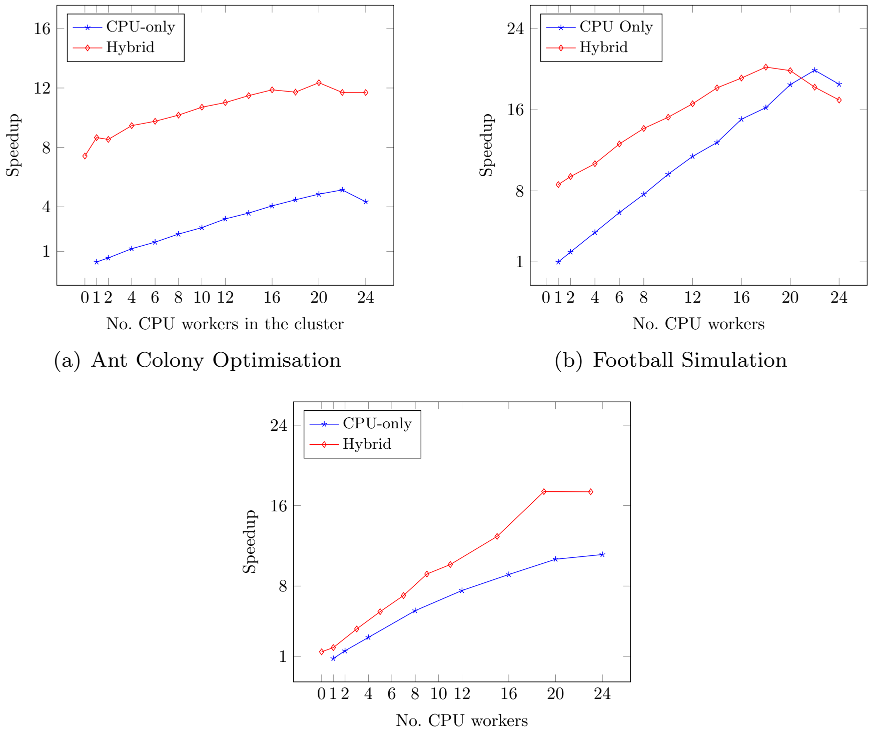
Ant Colony Optimisation (ACO) [6] is a heuristic for solving NP-complete optimisation problems. We apply ACO to the Single Machine Total Weighted Tardiness Problem (SMTWTP) optimisation problem, where we are given n jobs and each job, i, is characterised by its processing time, p_i deadline, d_i, and weight, w_i. The goal is to find the schedule of jobs that minimises the total weighted *tardiness*, defined as $\sum w_i \cdot \max\{0, C_i - d_i\}$, where C_i is the completion time of the job, i. The ACO solution consists of a number of iterations, where in each iteration each ant independently tries to improve the current best schedule, and is biased by a *pheromone trail.*The top-level skeleton structure is:

```
Pipe = {pipe, [{hyb_cluster, [{func, fun ant_c/1}], [{func, fun ant_g/1}]}],
                  TimeRatio, fun struct_size/1, fun make_chunk/2,
                  fun lists : flatten/1, NrCPUW, NrGPUW},
                      [{func, fun update_and_spawn/1}]},
Feedback = {feedback, [Pipe], fun ant_feedback/1},
```

A speedup graph is given in Figure 1(a). We can see that the CPU-only version shows modest speedups, up to 5.13 on 22 cores, degrading slightly when all 24 cores are used (this is probably the result of a saturation/scheduling conflict with other applications). The GPU-only version (where the number of CPU cores is 0) shows a better speedup of 7.52 than any CPU-only version. Combining CPU threads with a GPU gives clear benefits over either processor type alone, delivering speedups of up to 12.2 when 20 CPU threads and a GPU are used. The graph shows some anomalies, e.g. at 2 CPU threads plus a GPU, the performance is less than one CPU thread plus a GPU; at 18 CPU threads, there is a slight dip in speedup; and beyond 20 threads, performance plateaus. This can probably be mostly explained by the effects of work division described in Section 3.1. In the case of 1 CPU thread plus a GPU, our algorithm derives good work division where the CPU threads and GPU finish the execution at approximately the same time. Adding 1 thread, more work is given to the CPU threads, and in this case it may happen that the GPU finishes its portion of work earlier, resulting in imbalance.

4.2. Football Simulation

Football Simulation predicts the outcomes of football games, and is used by betting companies to calculate the winning odds for matches. Each simulation ac-



(a) Ant Colony Optimisation

(b) Football Simulation

(c) Image Merge

Figure 1. Speedups for Ant Colony, Football Simulation and Image Merge

cepts information about the teams involved in a match (e.g. attack and defence strength), and uses a randomised routine to predict the end result of the match. The final prediction for each match is averaged over multiple simulations. The top-level skeleton structure is

```
Cluster = {hyb_cluster, {func, fun(P) -> sim_match_cpu(P, NrSims) end},
                        {func, fun(P) -> sim_match_gpu(P, NrSims) end},
                        ChunkSizeMult, TimeRatio, NCPUW, NGPUW},
AllRes = skel:do(Cluster, AllPairs),
Results = [ get_average_score(OnePairRes) || OnePairRes <- AllRes ].
```

`P` is a pair of tuples that contains the necessary information about one match, i.e. information about one pair of teams. In the simplest case, we provide just two floating point numbers for each team, attack and defence strength. For each pair of teams, `sim_match_cpu` or `sim_match_gpu` is called `NrSimulations` times, and then the average score is computed using the `get_average_score` function. The speedups of *Football Simulation* are given in 1(b). Both CPU-only and hybrid version show improved speedups (from 1 and 8.3) when more CPU workers (and, therefore, CPU threads) are used, up to the point where the best speedup is obtained (20.2 with 18 CPU workers for hybrid version and 20 with 22 workers for CPU-only version). After this point, when more CPU workers are added, the performance starts to drop. This is due to unpredictability of performance of workers as the total number of Erlang processes approaches the number of cores, due to scheduling issues. This also has the effect that the division of work between CPU and GPU workers in the hybrid case is sub optimal when more than 18 CPU workers are used, explaining the earlier dip in performance for the hybrid case. Altogether, we can observe that when a smaller number of CPU threads are used, the hybrid version significantly outperforms the CPU-only version.

4.3. Image Merge

Image Merge is an application from the computer graphics domain. It reads a stream of pairs of images from files, and merges images from each pair. The top-level skeleton structure is

```
Farm = {hyb_farm, {func, fun merge_cpu/1}, {func, fun merge_gpu/1}},
FinalImages = skel:do(Farm, Images).
```

The speedups for Image Merge are given in Figure 1(c). We can observe that the hybrid version significantly outperforms the CPU-only version regardless of the number of CPU workers used, with the best speedups of 17 and 10 for hybrid and CPU-only version. We also observe an increase in speedup as the number of CPU workers increases. As usual, when the number of CPU workers approaches the number of CPU threads, we observe a drop in performance.

5. Conclusions and Future Work

This paper describes *Lapedo*, a system of hybrid skeletons for Erlang. Skeletons abstract commonly-used patterns of parallel computation, communication, and interaction into parameterised templates. Hybrid skeletons combine components that are specialised for different processor types , thus allowing efficient exploitation of heterogeneous multi-core/many-core systems while still offering a very high-level programming model. We have focused purely on CPU/GPU combinations, but since our library is built on top of OpenCL, it can also be used with other accelerators. We have also described a simple mechanism for dividing work between processor types. Finally, we have demonstrated *Lapedo* on three realistic Erlang applications. Our results show clear benefits of using hybrid skeletons, giving significantly better speedups compared to CPU-only skeletons with only a modest increase in programming effort, programmers are only required to write relatively-simple OpenCL kernels. In the future work, we plan to extend the *Lapedo* library with additional, domain-specific skeletons (e.g. orbit skeleton) and to adapt it to support emerging accelerator classes.

Acknowledgements

This work was funded by European Union FP7 grant “ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems” (code 288570) and H2020 grant “RePhrase: Refactoring Parallel Heterogeneous Resource-Aware Applications – a Software Engineering Approach” (code 644235). We would like to thank Christoph Kessler for very useful suggestions on improving the paper.

References

[1] J. Armstrong, S. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[2] I. Bozó, V. Fordós, Z. Horvath, M. Tóth, D. Horpácsi, T. Kozsik, J. Köszegi, A. Barwell, C. Brown, and K. Hammond. Discovering Parallel Pattern Candidates in Erlang. In *Proc. 13th Erlang Workshop*, Erlang '14, pages 13–23. ACM, 2014.

[3] C. Brown, M. Danelutto, K. Hammond, P. Kilpatrick, and A. Elliott. Cost-Directed Refactoring for Parallel Erlang Programs. *IJPP*, 42(4):564–582, 2014.

[4] M. I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation.* PhD thesis, 1988. AAID-85022.

[5] U. Dastgeer, C. Kessler, and S. Thibault. Flexible Runtime Support for Efficient Skeleton Programming. In K. de Bosschere, E. H. D'Hollander, G. R. Joubert, D. Padua, F. Peters, and M. Sawyer, editors, *Advances in Parallel Computing, vol. 22: Applications, Tools and Techniques on the Road to Exascale Computing*, pages 159–166. IOS Press, 2012. Proc. ParCo conference, Ghent, Belgium, Sep. 2011.

[6] M. den Besten, T. Sttzle, and M. Dorigo. Ant Colony Optimization for the Total Weighted Tardiness Problem. In *PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 611–620. 2000.

[7] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proc. HLPP '10*, pages 5–14. ACM, 2010.

[8] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, Nov. 2010.

[9] D. Grewe and M. F. OBoyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In J. Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 286–305. Springer Berlin Heidelberg, 2011.

[10] C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J. L. Träff, and S. Pllana. Programmability and Performance Portability Aspects of Heterogeneous Multi-/Manycore Systems. In *Proc. DATE-2012 conference on Design, Automation and Test in Europe, Dresden, Germany*, Mar. 2012.

[11] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[12] Rashkovskii, Yurii. Genomu: A Concurrency-Oriented Database. In *Erlang Factory SF*, 2013.

[13] Reed, Rick. Scaling to Milions of Simultaneous Connections. In *Erlang Factory SF*, 2012.

[14] T. Rogvall. OpenCL Binding for Erlang. `https://github.com/tonyrog/cl`.

[15] M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems. In *Par. Comp. Tech.*, Springer LNCS vol. 7979, pp. 258–272. 2013.

[16] Wilson, Ken. Migrating a C++ Team to Using Erlang to Deliver a Real-Time Bidding Ad System. In *Erlang Factory SF*, 2012.