

# Streaming Dynamic Coarse-Grained CPU/GPU Workloads with Heterogeneous Pipelines in FastFlow

Mehdi Goli, Michael T. Garba, Horacio González-Vélez  
*IDEAS Research Institute, Robert Gordon University, Aberdeen, Scotland, UK*  
*Email: {m.goli,m.t.garba,h.gonzalez-velez}@rgu.ac.uk*

**Abstract**—Software pipelines permit the decomposition of a repetitive sequential process into a succession of distinguishable sub-processes called stages, each of which can be concurrently executed on a distinct processing element. This paper presents a heterogeneous streaming pipeline implementation using the FastFlow skeletal library for a numerical linear algebra code. By introducing minimal memory management, we implement a large-scale streaming application which allocates the different pipeline stages to multi-core CPU and multi-GPU resources in a cluster environment, demonstrating the suitability of the algorithmic skeleton approach to efficiently coordinate the pipeline operation. Our implementation shows that long-running heterogeneous pipelines can be effectively implemented in FastFlow.

**Keywords:** Algorithmic Skeletons; Parallel Patterns; GPU; Parallel Programming; Cluster Computing;

## I. INTRODUCTION

Efficient heterogeneous parallel computing with CPUs and GPUs remains an open area of study due to the intrinsic complexity and potential trade-offs associated with the communication and non-linear performance characteristics that exist in GPU-accelerated nodes despite the presence of high performance interconnects. Current trends indicate that, with major hybrid architectures in development, techniques and tools for managing these heterogeneous platforms are of increasing significance.

Algorithmic skeletons (or simply skeletons) have long been considered a viable approach to introduce high-level abstraction to parallel programming that hides the complexity of recurring patterns of coordination and communication logic behind a generic reusable application interface [1], [2].

In particular, pipelines enable the decomposition of a repetitive sequential process into a succession of distinguishable sub-processes called stages, each of which can be concurrently executed on a distinct processing element. Pipelines are exploited at coarse-grained level in parallel applications employing multiple processors. Coarse-grained pipelines refine complex algorithms into a sequence of independent computational stages where the data is “piped” from one computational stage to another. Each stage, composed by a simple consumer, a computational function, and a simple producer, is then allocated to a processing element in order to compose a parallel pipeline.

When handling a large number of streaming inputs, it is throughput rather than latency which constrains overall efficiency, since the latency is only relevant to measure the time to fill up the pipeline initially. Once at capacity, the pipeline steadily delivers results at the throughput ratio. Hence, in order to improve the overall efficiency of a parallel pipeline, it is necessary to minimise the bottleneck processing time, potentially allocating the distinct stages to elements of dissimilar architectures (CPU/GPU).

The FastFlow framework is a C++ shared memory algorithmic skeleton implementation developed for cache-coherent multi-core architectures. Having demonstrated better performance than Intel’s TBB library and OpenMP implementations for some pertinent programs [3], [4], we hypothesise that its skeletal pipeline can be of use to coordinate a computationally-intensive application on a tightly-coupled heterogeneous multicore CPU/GPU cluster architecture.

### A. Contribution

The FastFlow framework has been extensively tested in shared-memory multi-core environments, however integrated GPU support within the libraries remains under development. Scant research has been conducted to evaluate the suitability of FastFlow for workloads that incorporate GPU kernels. Furthermore, the performance characteristics and stability for large-scale resource-heavy parallel computing applications remain untested.

In this paper, we demonstrate an implementation of a scalable GPU numerical linear algebra testbed as a streaming pipeline using the FastFlow framework to coordinate dynamic work distribution between multi-core CPU and GPU resources. Our implementation shows that, with the introduction of minimal memory management, long-running heterogeneous pipelines are readily implemented in FastFlow.

This paper is organised as follows. In section II, we briefly describe the background and FastFlow architecture. In section III we explain the proposed approach for a pipeline implementation in FastFlow and propose a solution to the memory management problem that emerges. Section IV presents the experimental infrastructure used for evaluating the results, followed by the result of our evaluation. Finally,

section V provides some concluding remarks and future work.

## II. BACKGROUND

Following earlier work in developing a parallel implementation of INS codes for traditional multicore and multinode architectures [5], we have developed an implementation of the relevant linear algebra routines from EISPACK targeting GPUs and using the CUDA platform. However, architectural challenges place constraints on the integration of these high performance GPU kernels with the original SPMD implementation. These constraints highlight that, even for massively parallel applications, traditional techniques of structuring parallel programs may be incompatible with the specific performance requirements and runtime characteristics introduced by heterogeneous GPU accelerators [6]. These issues include:

- significant application memory demand;
- significant memory transfer costs between host main memory and GPU device memory;
- page-locking restrictions on large regions of memory that may be required for asynchronous data transfer [7];
- alternate idling of both CPU and GPU resources; and,
- reliance of GPU performance on availability of sufficient work to keep all streaming multiprocessors active.

These may be considered limitations of the conventional view of GPUs as merely accelerators. On the one hand, in seeking techniques to optimally balance the utilisation of heterogeneous resources dynamically, previous work in the field suggests that a parallel pipeline pattern [8], [9] is a viable approach to minimising CPU and GPU idle times, whilst exposing a high-level application-independent interface to the application programmer and taking advantage of the intrinsic pipeline characteristics of the GPU hardware [10]. While it has been demonstrated that a polynomial time solution exists for identical processors in a linear pipeline, introducing heterogeneity creates an NP-hard allocation problem [11].

On the other hand, traditional GPU approaches to computational linear algebra problems have been tailored to the architecture using a particular mapping of matrices rows and columns [12], a translated representation to store matrices as texture maps and then exploit pixel-based functions to implement arithmetic operations [13], or by optimising CUDA kernels [14]. While the more recent MAGMA library [15] has aggregated a significant repository of GPU-optimised functions for linear algebra, it has not taken a pattern-based/skeletal approach to decouple the coordination from the computation of the overall resulting application, a shared view that has started to gain momentum among the functional parallel programming community [16].

## FastFlow

FastFlow is a parallel programming framework for multicore platforms which facilitates the development of shared memory parallel applications by providing abstraction layers, programming constructs and composable algorithmic skeletons [4].

As shown in figure 1, FastFlow provides three layers of abstraction and building blocks between the underlying multicore/manycore hardware and the application:

- 1) **Simple Streaming Networks** which are basically lock-free Single-Producer-Single-Consumer (SPSC) queues.
- 2) **Arbitrary Streaming Networks** which are generalised Single-Producer-Single-Consumer (SPSC) queues that provide one-to-many (SPMC), many-to-one (MPSC), and many-to-many (MPMC) queues. Dataflow and synchronisation between queues are transparently handled.
- 3) **Streaming Network Patterns** such as the farm and pipeline skeletons. Arbitrarily nested or sequential combinations are possible.

FastFlow streaming patterns are coordinating mechanisms that control the flow of work between multiple concurrent threads. This frees programmers to focus on the application-specific computation aspect. For existing legacy codes, FastFlow further provides *FastFlow-Accelerator*, allowing users to move or copy parts of sequential codes into the body of C++ methods, for parallel execution in a FastFlow skeleton. When suitable, this may reduce programming effort and allow the application to exploit previously unused cores in a scalable manner. The cache coherent characteristics have been reported to deliver better performance than alternative implementations of fine-grained parallelism with Intel's Thread Building Blocks and OpenMP [4].

The skeleton approach and the abstraction of complex coordination and communication logic is particularly promising in the context of heterogeneous multicore CPU and GPU platforms. However, as support for GPUs remains under development, the suitability of FastFlow for problems of this kind has not been totally established. Furthermore, the existing application domain of FastFlow has consisted primarily of fine-grained parallelism. We are interested in exploring the behaviour of this framework for *large scale* coarse-grained workloads that may only be practical on dedicated (potentially distributed) clusters. This places unprecedented demands on the FastFlow architecture and tests the maturity of its implementation.

Therefore, the novelty of our approach lies in the exhaustive performance evaluation of a large-scale hybrid GPU/CPU computational linear algebra code using the FastFlow environment.

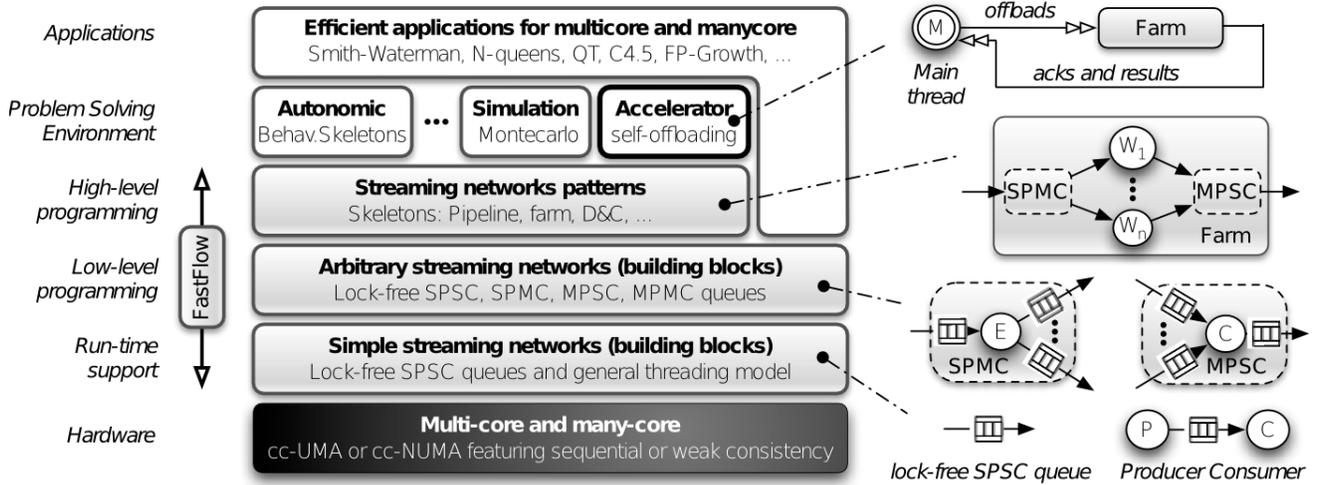


Figure 1. FastFlow Architecture.

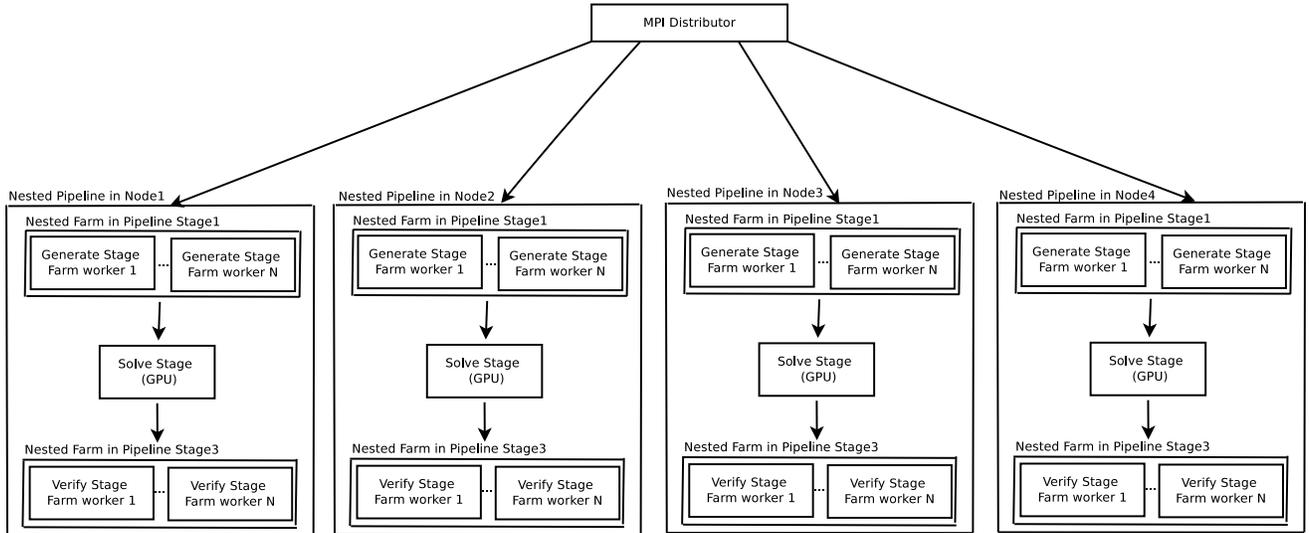


Figure 2. A three-level hierarchy of skeletons, consisting of a distributed farm of MPI workers at the top-level containing nested pipeline workers with nested farm stages and a GPU stage.

### III. A HETEROGENEOUS CPU/GPU PIPELINE IN FASTFLOW

A practical problem is validation of our implementation of the EISPACK routines for solving Hermitian eigensystems in a setting that bears architectural similarity to the final intended deployment and presents similar computational demand. To solve this, we implement a pipeline using FastFlow constructs. The three pipeline stages involve:

- 1) **Generation** of suitable Hermitian test matrices,  $A$ .
- 2) **Solution** using the test GPU kernels to compute the eigenvectors  $E$  and eigenvalues  $D$  on the GPU.
- 3) **Verification** of the computed eigenvectors and eigenvalues on the CPU. For a matrix of eigenvectors  $E$

and a corresponding diagonal matrix of eigenvalues  $D$ , we expect that  $AE = ED$ . Therefore, a possible error function is the Frobenius norm of the matrix

$$\epsilon = \|AE - ED\|_F$$

where the Frobenius norm of an  $m \times n$  matrix  $M$  is defined as

$$\|M\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |m_{ij}|^2}$$

This presents a massively parallel computational problem that is readily implemented as a three-stage pipeline. The

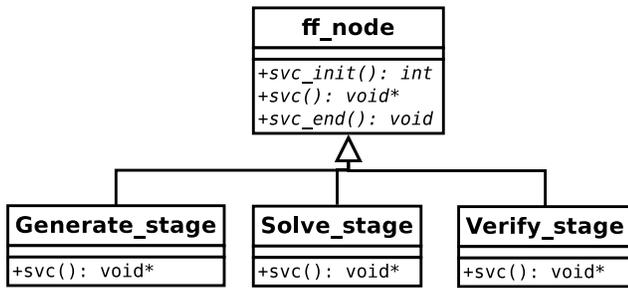


Figure 3. UML notation of pipeline stages and their dependencies.

pipeline stages exchange batches of work based on the optimal requirements of the GPU solution stage where the performance is reliant on the CUDA grid size at kernel launch.

Generation and verification in the first and final stages of the pipeline operate on these batches where the individual problems are independent, this presents another level of parallelism that is well suited to the farm skeleton. Employing skeletal composition allows a nested parallel hierarchy.

FastFlow eases the implementation by providing the pipeline coordination construct. A prototype class is provided with three virtual overridable member functions:

- `svc_init`
- `svc`
- `svc_end`

The `svc` method represents the actual computation performed in each stage. The prototype class, derived from the `ff_node` class, may represent either a generic pipeline stage or worker in a farm.

We subclass `ff_node` to implement a `Generate_stage`, `Solve_stage` and `Verify_stage` representing the initial, intermediate and terminal stages of the pipeline (Figure 3). The `Solve_stage` invokes our external GPU-enabled linear algebra libraries, as the current version of FastFlow does not provide any direct mechanisms for managing the execution GPU kernels. The pipeline stages themselves are added to a `ff_pipeline` container object in the proper sequence and execution is started by calling the `ff_pipeline::run_and_wait_end` method (Figure 4).

At present, FastFlow executes each instance of `ff_node` as a single thread. However, the individual pipeline stages may be amenable to further parallelisation. To overcome this limitation, arbitrary skeleton nesting is possible within each instance of `ff_node`. In our implementation, we found it difficult to justify the conceptual complexity of nesting the FastFlow farm skeleton within the initial and final stages of the pipeline. In order to achieve simple multi-threaded functionality, we resorted to direct use of OpenMP to implement the farm-worker pattern.

Another limitation is that FastFlow provides no distributed memory primitives, thus, we create an additional level of

```

1 int main(int argc, char * argv[])
2 {
3     ...
4     ff_pipeline pipe;
5     ff_node *generate_stage;
6     ff_node *solve_stage;
7     ff_node *verify_stage;
8
9     //Instantiate pipeline stages
10    generate_stage= new Generate_stage();
11    solve_stage = new Solve_stage();
12    verify_stage = new Verify_stage();
13
14    //Add stages to pipeline
15    pipe.add_stage(generate_stage);
16    pipe.add_stage(solve_stage);
17    pipe.add_stage(verify_stage);
18
19    ...
20
21    //start the pipeline
22    pipe.run_and_wait_end();
23
24    ...
25 }
  
```

Figure 4. Instantiating and running the pipeline .

nesting using an MPI farm to handle distribution over multiple nodes in a cluster. This three-level hierarchy, consisting of a farm of nested pipeline workers with nested farm stages is illustrated in Figure 2.

While FastFlow has been evaluated on small-scale and fine-grained parallel jobs, large-scale, coarse grained jobs with significant memory demands are untested. Early testing revealed that the pipeline implementation in FastFlow leads to steadily increasing buffer levels before bottleneck stages. For long-running high-throughput pipelines, this very rapidly consumes all available memory on the machine and leads to a fatal program error.

With minimal modifications to the underlying framework, we have introduced adaptive pipeline throttling to regulate the memory footprint of the application and maintain usage within configurable bounds. This feature is particularly significant in streaming applications where processing occurs outside FastFlow and it is necessary to provide sufficient memory for other processes. The pipeline is starved of input when the memory usage exceeds the `STOP_THRESHOLD` as a percentage of total memory. Processing on the other stages proceeds until memory usage falls below the `START_THRESHOLD`, at this point, the initial stage thread is awakened and new input is injected into the pipeline (Figure 5). The conceptual simplicity and effectiveness of this approach makes it an attractive solution to the memory management problem. Processing continues as other parallel pipeline stages make use of the available hardware, regardless of the actual location of the bottleneck stage.

```

1  /* if memory is low, go to sleep */
2  void pushwait(){
3
4      pthread_mutex_lock( &count_mutex );
5
6      if (get_free_ram() < STOP_THRESHOLD){
7          pthread_cond_wait( &condition_var,
8                          &count_mutex );
9      }
10
11     pthread_mutex_unlock( &count_mutex );
12
13 }
14
15
16 /* if sufficient memory has been freed wake the
17    sleeping stages */
18 void popsignal(){
19
20     pthread_mutex_lock( &count_mutex );
21
22     if (get_free_ram() > START_THRESHOLD){
23         pthread_cond_signal( &condition_var );
24     }
25
26     pthread_mutex_unlock( &count_mutex );
27
28 }

```

```

1  void* Generate_stage::svc(void * t) {
2
3      ...
4
5      pushwait();
6
7      task_t *task = generate(...);
8
9      return (void*)task;
10
11 }
12
13 }
14
15 void* Verify_stage::svc(void * task_in) {
16
17     ...
18     task_t *task = (task_t*)task_in;
19
20     verify(task);
21
22     popsignal();
23
24     return task;
25
26 }

```

Figure 5. The left side code represents the functions provided for the memory management problem and the right side code shows the invocation of each function inside the stages.

Table I  
CLUSTER HARDWARE SPECIFICATION.

Parameter	Value
No. of cluster nodes	4
CPUs per Node	2
Cores per CPU	6
CPU Clock	3.07 GHz
Physical Memory per Node	6 GB
GPUs per Node	1
GPU Model	NVIDIA Tesla M2090
GPU Memory	6 GB
CUDA Cores	512
CUDA Version	4.0 V0.2.1221
CUDA Driver Version	290.10

#### IV. PERFORMANCE EVALUATION

Performance evaluation was carried out on a 4-node multi-GPU test cluster (hardware specifications in Table I) with 4 MPI processes, and 12 workers at each nested farm in the pipeline stages. 55,296 pseudorandom Hermitian test matrices of order 1024 and at double precision were streamed through the pipeline to establish that the computed error is within acceptable tolerance.

Table II indicates that while the total application runtime on the cluster was 9227 seconds (or 2.53 hours), all GPUs in the cluster were computationally active for a minimum of 8930 seconds (or 2.48 hours). This represents good overlap

Table II  
ACTIVE EXECUTION TIMES ON INDIVIDUAL GPUS AND TOTAL PROGRAM RUNTIME ON TEST CLUSTER.

Resource	Runtime (seconds)
GPU1	8995.6
GPU2	8930.1
GPU3	8930.7
GPU4	8995.4
Total Cluster Runtime	9227.0

of the CPU and GPU stages of the pipeline, validating the choice of this architectural style for maximising resource utilisation.

Figure 6 presents memory usage over one hour of execution for the entire cluster. Set at 50% of the total 200 GB of physical memory available, memory usage remains within the pre-configured limits with new work units injected into the pipeline when free memory rises above 100 GB and throttling enforced below 80 GB. It is evident that the memory demands of this application are substantial.

As expected, the GPU stage constitutes the primary bottleneck to the pipeline. Figure 7 indicates that CPU usage varies between 30% and 60% following variations in the number of active processes.

The use of adaptive memory management is crucial for allowing FastFlow to scale up to large problems. These measures are also suited to machines with constrained main

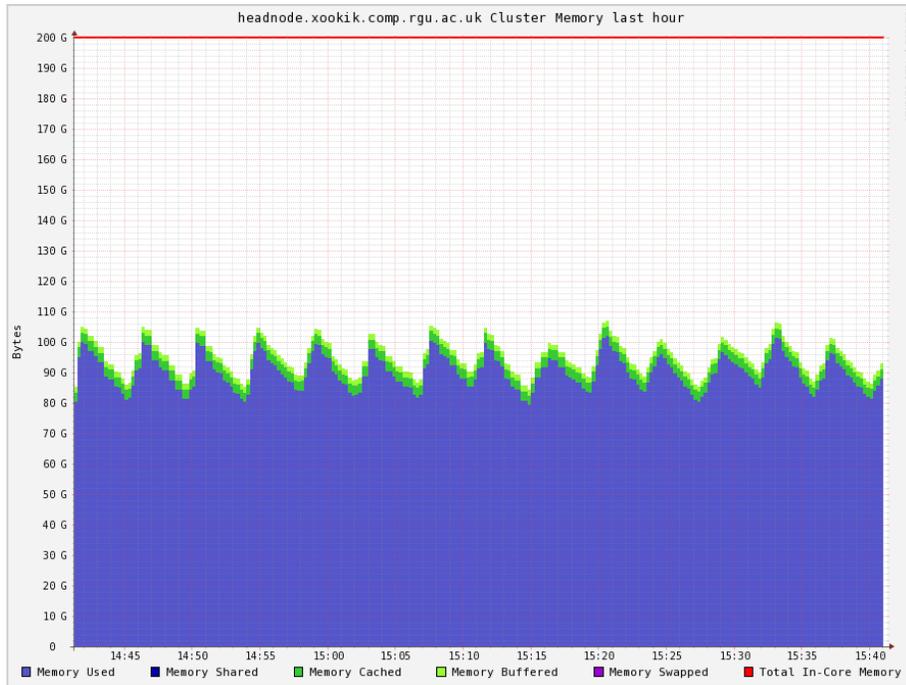


Figure 6. Overall memory usage over one hour of execution for all four cluster nodes. STOP\_THRESHOLD and START\_THRESHOLD are respectively 50% and 40% of the total 200GB available physical memory. The distinctive saw-tooth waveform follows from intermittent throttling of the pipeline.

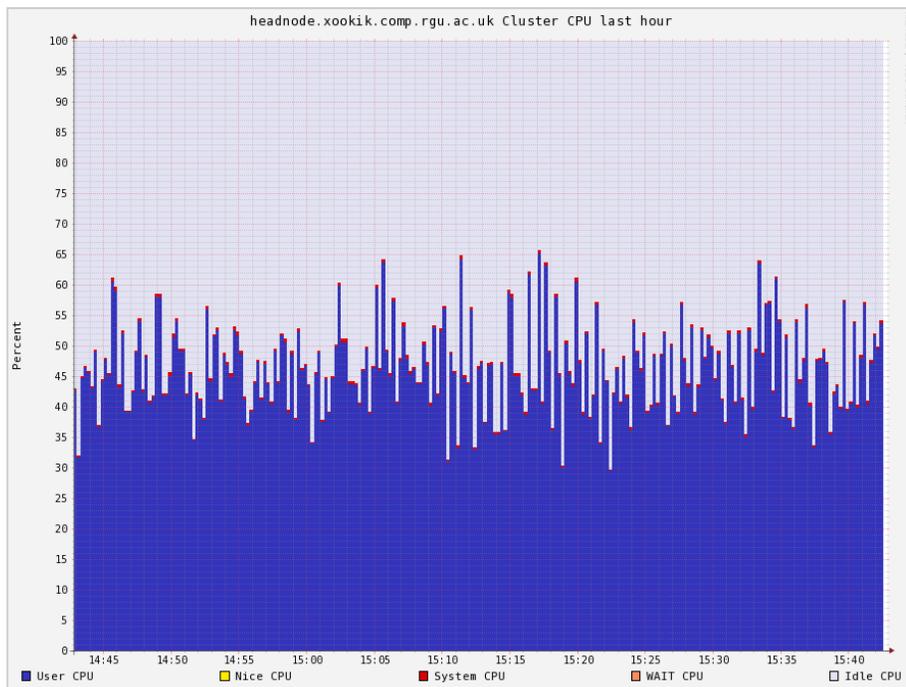


Figure 7. Overall cluster CPU usage percentage over one hour of execution. As the GPU stage constitutes the bottleneck, throttling adaptively imposes a limit on CPU usage, preventing pipeline queue overflows.

memory - as was the case with our original development platform. Moreover, the close correspondence between active GPU computing times (Table II) and the total application runtime is an indicator of near-ideal GPU utilisation.

## V. CONCLUSION AND FUTURE WORK

In this paper we have applied FastFlow to an unprecedented large scale computational problem. The stability of FastFlow while scaling to a problem of this size attests to the generality of the design and the strengths of the algorithmic skeleton approach. However, it is clear that adaptive memory management schemes are necessary for frameworks of this kind to meet their full potential for high-throughput, coarse-grained and resource intensive workloads.

Other limitations exist. Without distributed memory support in FastFlow, we find that an MPI implementation layer is necessary to achieve execution on a cluster. One aspect of the appeal of the skeleton concept is that the developer can potentially be insulated from the intricacies of particular libraries and implementation issues that arise. Furthermore the stages in the pipeline skeleton operate as single worker threads, a useful option would be a configurable pool of workers threads mapped to pipeline stages. Again, our implementation overcomes this limitation by employing skeleton composition and creating nested farms within these stages.

The FastFlow authors have indicated their interest in heterogeneous execution environments. In theory, the ability to incorporate arbitrary functions in FastFlow allows the use of GPU resources with existing kernels. Practically, GPUs present additional challenges in terms of the wide variation in capabilities, performance and efficiency constraints. A particularly promising prospect would be the availability of a direct GPU back-end in the framework that transparently provides hybrid CPU/GPU execution of certain functions. We recognise that this is not a trivial problem.

In the future, we intend to implement adaptive memory management in cooperative execution modes where other processes may be present on the same node.

Earlier attempts at using the FastFlow farm for the GPU stage resulted in a fatal system error as a large number of workers attempted to simultaneously use the GPU. There may be a potential to impose resource usage management on FastFlow farms without altering the underlying semantics of the skeleton. This should allow the farm skeleton to function in an environment where there is resource contention but no explicit dependency between the individual workers in the farm.

## ACKNOWLEDGMENT

This work has been partially funded by the collaborative project *ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems* <http://paraphrase-ict.eu/>, funded by the European Commission Seventh Framework

Programme Subprogramme area: ICT-2011.3.4 Computing Systems under contract no.: 288570 (10/2011-9/2014).

## REFERENCES

- [1] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, ser. Research Monographs in Parallel and Distributed Computing. London: MIT Press/Pitman, 1989.
- [2] H. González-Vélez and M. Leyton, “A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers,” *Software–Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.
- [3] M. Aldinucci, M. Meneghin, and M. Torquati, “Efficient Smith-Waterman on multi-core with FastFlow,” in *PDP 2010*. Pisa: IEEE, Feb. 2010, pp. 195–199.
- [4] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, “Accelerating code on multi-cores with FastFlow,” in *Euro-Par 2011*, ser. LNCS, vol. 6853. Bordeaux: Springer, Aug. 2011, pp. 170–181.
- [5] M. Garba, H. González-Vélez, and D. Roach, “Parallel computational modelling of inelastic neutron scattering in multi-node and multi-core architectures,” in *IEEE HPCC-10*. Melbourne: IEEE, Sep. 2010, pp. 509–514.
- [6] M. T. Garba and H. González-Vélez, “Asymptotic peak utilisation in heterogeneous parallel CPU/GPU pipelines: A decentralised queue monitoring strategy,” *Parallel Processing Letters*, 2012, to appear.
- [7] NVIDIA Corporation, “NVIDIA CUDA C Programming Best Practices Guide,” NVIDIA, Santa Clara, CA 95050, USA, Manual Version 2.3, 2009, available from: <http://developer.nvidia.com/> (Last Accessed: 1 Feb 2012).
- [8] F. Almeida, D. Gonzalez, L. M. Moreno, and C. Rodriguez, “Pipelines on heterogeneous systems: models and tools,” *Concurrency and Computation: Practice and Experience*, vol. 17, no. 9, pp. 1173–1195, 2005.
- [9] H. González-Vélez and M. Cole, “Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms,” *Concurrency and Computation–Practice and Experience*, vol. 22, no. 15, pp. 2073–2094, 2010.
- [10] D. Luebke and G. Humphreys, “How GPUs work,” *Computer*, vol. 40, no. 2, pp. 96–100, Feb. 2007.
- [11] A. Benoit, A. Dobrila, J.-M. Nicod, and L. Philippe, “Workload balancing and throughput optimization for heterogeneous systems subject to failures,” in *Euro-Par 2011*, ser. LNCS, vol. 6853. Bordeaux: Springer, Aug. 2011, pp. 242–254.
- [12] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, “LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware,” in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC ’05. Seattle: IEEE Computer Society, 2005, pp. 3–.
- [13] J. Krüger and R. Westermann, “Linear algebra operators for GPU implementation of numerical algorithms,” in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH ’05. Los Angeles: ACM, 2005.
- [14] V. Volkov and J. Demmel, “Benchmarking GPUs to tune dense linear algebra,” in *SC 2008*, Austin, Nov. 2008, pp. 1–11.
- [15] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, “Optimizing symmetric dense matrix-vector multiplication on GPUs,” in *SC ’11*. Seattle: ACM, 2011, pp. 6:1–6:10.
- [16] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, “Accelerating Haskell array codes with multicore GPUs,” in *DAMP ’11*. Austin: ACM, 2011, pp. 3–14.