Algorithmic Skeleton Framework for the Orchestration of GPU Computations*

Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros

CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2829-516 Caparica, Portugal
herve.paulino@fct.unl.pt

Abstract. The Graphics Processing Unit (GPU) is gaining popularity as a co-processor to the Central Processing Unit (CPU). However, harnessing its capabilities is a non-trivial exercise that requires good knowledge of parallel programming, more so when the complexity of these applications is increasingly rising. Languages such as StreamIt [1] and Lime [2] have addressed the offloading of composed computations to GPUs. However, to the best of our knowledge, no support exists at library level. To this extent, we propose Marrow, an algorithmic skeleton framework for the orchestration of OpenCL computations. Marrow expands the set of skeletons currently available for GPU computing, and enables their combination, through nesting, into complex structures. Moreover, it introduces optimizations that overlap communication and computation, thus conjoining programming simplicity with performance gains in many application scenarios. We evaluated the framework from a performance perspective, comparing it against hand-tuned OpenCL programs. The results are favourable, indicating that Marrow's skeletons are both flexible and efficient in the context of GPU computing.

1 Introduction

The GPU has been maturing into a powerful general processing unit, surpassing even the performance and throughput of multi-core CPUs in some particular classes of applications. The GPU architecture is specially tailored for data-parallel algorithms, where throughput is more important than latency. This makes them particularly interesting for high-performance computing on a broad spectre of application fields [3]. However, the base parallel computing frameworks for General Purpose Computing on GPUs (GPGPU), CUDA [4] and OpenCL [5], require in-depth knowledge of the underlying architecture and of its execution model, such as the disjointness of the host's and the device's addressing spaces, the GPU's memory layout, and so on. Consequently, high-level

* This work was partially funded by FCT-MEC in the framework of the PEst-OE/EEI/UI0527/2011 - Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) - 2011-2012 and project PTDC/EIA-EIA/102579/2008 - Problem Solving Environment for Materials Structural Characterization via Tomography.

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 874–885, 2013.

GPU programming is currently a hot research topic that has spawned several interesting proposals, e.g. OpenACC [6], Lime [2], Chapel [7] and StreamIt [1].

Nonetheless, as software developers become increasingly familiarised with GPU computing, and the overall hardware computational power is consistently growing, the complexity of GPU-accelerated applications is tendentiously higher. This *status quo* raises new challenges, namely how to efficiently offload this new class of computations to the GPU without overburdening the programmer. High-level programming frameworks will have to split their focus between the generation of OpenCL or CUDA kernels from higher level constructs, and the generation of the orchestration required on the host side. Languages such as StreamIt and Lime expand the use of the GPU beyond the usual offloading of a single kernel, offering more sophisticated constructs for streaming, pipelining and iterative behaviours. However, the impact of these constructs is restrained by the adoption of a new programming language.

To this extent, our proposal is to provide this expressive power at library level. For that purpose, we build on the concept of algorithmic skeleton to propose a framework for orchestrating the execution of OpenCL kernels that offers a diverse set of compoundable data- and task-parallel skeletons. The generation of OpenCL kernels from source code, namely C++, is orthogonal to this work and can grow from existing tools, such as the Offload compiler [8].

To the best of our knowledge, SkePU [9], SkelCL [10] and Muesli [11] are the sole Algorithmic Skeleton Frameworks (ASkFs) to address GPGPU. Nonetheless, all of them focus on the high-level expression of simple GPU computations, hence supporting only variants of the *map* skeleton that apply a user-defined function.

The contributions of this paper are therefore: 1 - the *Marrow* C++ ASkF for the orchestration of OpenCL computations (Section 3). Marrow pushes the state of the art by extending the set of GPU supported skeletons, introducing ones such as *pipeline*, *stream*, and *loop*, and by allowing these to be nested, thus providing a highly flexible programming model. Moreover, it is optimized for GPU computing, introducing transparent performance optimizations, specifically through a technique known as overlap between communication and computation. 2 - A comparative performance evaluation against OpenCL, whose experimental results attest the quality of our prototype implementation (Section 4).

2 Related Work

Skeletons are a high level parallel programming model that hide the complexity of parallel applications, by implicitly performing all the non-functional aspects regarding parallelization (e.g., synchronization, communication). Additionally, basic skeletons may be combined (nested) into more complex constructs, adding structural flexibility to the application. Skeletons are usually made available as algorithmic skeleton frameworks, analogous to common software libraries.

SkePU [9], SkelCL [10] and Muesli [11] are the only ASkFs to address GPU computing, using C++. They have many common features, focused solely on data-parallel skeletons. A Vector concept is used in all three to abstract data

operations, and to introduce implicit optimizations, such as lazy copying. This feature postpones data transfers until needed, and allows different skeletons to access the same data without transferring it back to host memory. None of these ASkFs support skeleton nesting, thus no compound behaviours can be offloaded to the GPU. SkePU offers five skeletons: *map*, *reduce*, *map-reduce*, *map-overlap*, and *map-array*. The programmer can decide which is the target execution platform (CPU or GPU) at compilation time, as SkePU supports both CUDA and OpenCL. Skeleton behavior is specified using a set of predefined macros, although, macro compatibility is not universal among all skeletons. SkelCL supports four basic skeletons: *map*, *reduce*, *zip*, and *scan*. It generates OpenCL code from the aggregation of user-defined functions (supplied as strings) and predefined skeleton code. Muesli is a template library which supports both clusters, multi-core CPUs (OpenMP) and GPUs (CUDA). Among the multiple skeletons it supports, only *fold*, *map*, *scan* and *zip* are allowed on the GPU. More recently, AMD has announced Bolt [12], a C++ template library that provides OpenCL-accelerated *sort*, *transform* (similar to map) and *reduce* patterns. Data is defined through a Vector concept akin to the previous ASkFs, whist user-defined functions are defined through macros encasing a structure with C++ code.

The Lime [2] and StreamIt [13] programming languages provide primitives close to the ones we are proposing in this work, such as the *pipeline*. StreamIt is a stream processing language that enables the creation of disciplined graphs by combining three kinds of constructs: *pipeline*, *split-join* and *feedback-loop*. Recently it has been equipped with the CUDA back-end [1]. All the GPU execution is generated by the compiler, and optimized using a profiling stage. Lime is a language that supports kernel offloading and streaming constructs (akin to StreamIt) using a pipeline operator, while maintaining compatibility with the Java language. Data serialization is required to interface Java with C, presenting a considerable drawback, as data transfers become expensive.

When compared to our proposal, these languages have a narrower scope, as complex GPU applications are limited to the algorithms effectively compatible with the streaming programming model. As Marrow supports the offloading of complex skeletons compositions and streaming constructs, it breaks away from the current ASkF's data-parallel skeletons. This enables the usage of any GPU computation regardless of the applications' remaining purposes.

3 The Marrow Algorithmic Skeleton Framework

Marrow is a dynamic, efficient, and flexible C++ ASkF for GPGPU. We suggest constructs that orchestrate most of the details resulting from the underlying programming model, leaving the developer to focus on the parallel computations (kernels). The latter are common OpenCL kernels orchestrated by the skeletons to achieve a particular parallel application schema.

We are primary interested in skeletons whose execution behaviour is not hampered by the logical, and physical, division between the host and device address spaces. Skeletons whose execution is based on persistent data schemes are particularly attractive, since they do not require data transfers when combining

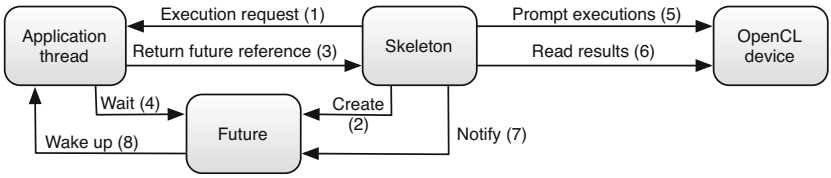

Fig. 1. Marrow's execution model

distinct execution instances. In this way, they avoid the overheads associated to transfers between disjoint memory spaces. Consider, for example, a *pipeline*; in a GPU execution the data produced by a stage i does not have to be transferred back to main memory in order to be available to the subsequent stage $(i+1)$. On the other hand, we target skeletons that provide functionalities that are useful in the usual GPGPU utilization domains. For instance, a *loop* skeleton for iterative computations is of particular interest to the scientific computing field.

We deem as fundamental the nesting of skeletons, as it enables the construction of complex executional structures, possibly containing distinct behaviours, in a simple and efficient manner. This technique is also beneficial performance-wise, in the sense that it is compatible with a disjoint memory layout. An application may apply a successive collection of computations, in the form of skeletons, to an input dataset, and only carry out memory transfers when: writing the input to device memory, and reading the results to main memory. Furthermore, the nesting mechanism embeds the construction of complex structures, allowing the skeletons to focus simply on particular functional behaviours.

Lastly, we seek to introduce transparent performance optimizations by taking advantage of the possibility to overlap communication with computation. By doing so, the skeletons can make better use of the features of modern GPUs, and increase overall efficiency. The transparent application of such optimization enables the development of efficient GPU accelerated applications without a large degree of knowledge of both parallel programming and OpenCL orchestration.

3.1 Execution Model and API

Marrow's execution model, depicted in Figure 1, promotes the decoupling of the skeleton computations from application execution. Given that GPU executions stretch through a somewhat extended period of time, it makes sense to free up the application to perform additional computations while the device carries out its task. This execution model can be regarded as master/slave pattern, on which the application offloads asynchronous executions requests. The submission of such a request to a skeleton (step 1 in the figure) is not immediately relayed to the device . Instead, it is queued, an associated *future* object is created (step 2) and its reference returned to the application (step 3). The *future* allows the application to, not only, query the state of the execution, but also wait until the results are ready (step 4). As soon as the skeleton becomes available to fulfil the execution request, it performs the necessary orchestration to properly carry

out the desired computation on the device (step 5). Subsequently, once the results are read to the host memory (step 6) the respective future is notified (7), which, in turn, may wake up the submitting application thread (step 8).

This execution model motivates a rather simple API. Issuing a skeleton execution is accomplished through an asynchronous `write` operation that requests an OpenCL execution, and renders a future object.

3.2 Nesting

A nested skeleton application can be regarded as a composed acyclic graph (composition tree), on which every node shares a general computational domain. Each of these nodes can be categorized according to the interactions with its ancestor/children, the resources it manages, and the types of operations it issues to the device. The categories are: *root node*, *inner node*, and *leaf node*.

The root node is the primary element of the composition tree. It is responsible for processing the application's execution requests, which naturally implies submitting one or more OpenCL executions. Therefore, it must manage most of the resources necessary to accomplish such executions, as well as performing data transfers between host and device memory. Additionally, it prompts executions on its children, parametrizing them with a specific set of resources, e.g. the objects on which they must issue executions, or the memory objects that must use as input and output.

Inner nodes are skeletons whose purpose is to introduce a specific execution pattern/behaviour to their sub-tree. These nodes might not need to allocate resources, since they are encased in a computational context created by the root, but resort to that same context to issue execution requests on their children.

Leaf nodes should not be referred to as skeletons because they export an executional entity, rather than introducing a specific execution pattern. Consequently, they are represented by `KernelWrapper` objects that encapsulate OpenCL kernels, and are used to finalize the construction of the composition tree.

To be compatible with the nesting mechanism, i.e. become an inner-node, a skeleton must be able to perform its execution on pre-initialized device memory, issued by its ancestor. Furthermore, it should be able to share an execution environment with other nodes, even if it adds state (e.g., memory objects, executional resources) to that environment. By contrast, a skeleton whose executional pattern requires the manipulation of input/output data on host memory is incompatible with the nesting mechanism, and thus can only be used as a root node. In any case, a skeleton that supports nesting is also eligible to become the root of a composition tree.

Implementation: One of the main challenges imposed by skeleton nesting is the standardization of distinct computational entities, in a manner that provides a cross-platform execution support. The solution must abstract a single entity from the particularities of others, and yet, provide a simple and well defined invocation mechanism. Furthermore, there are issues intrinsic to GPU computing. For instance, the efficient management of the platform's resources craves

the sharing of OpenCL structures - command-queues, memory objects, and the context (upon which skeletons may allocate resources) - between the skeletons that build up a composition tree.

To tackle these issues, skeleton nesting in Marrow is ruled by the `IExecutable` interface that must be implemented by every nestable skeleton, as well as `KernelWrapper`. The interface specifies a set of functionalities that together enable a multi-level nestable execution schema. These include some core requisites, such as the sharing of execution contexts, the convey of inner-skeleton executions, or even the provisioning of executional information to a node's ancestor.

Another challenge is the management of node communication and synchronization. Even though Marrow's execution flow runs downward, from the root to the leafs, nodes at different heights must be synchronized so that their application correctly follows the defined execution pattern. For instance, a node with two children has to ensure that these are prompted in the right order, and that each has its data dependencies solved before being scheduled to execute. Internode communication must also be taken into account, since it is vital that the nodes read from, and write to, the appropriate locations. Ergo, both synchronization and communication are seamlessly supported by the nesting mechanism, allowing skeletons to perform these complex tasks in the simplest way possible.

3.3 Overlap between Communication and Computation

Overlap between communication and computation is a technique that takes advantage of the GPU's ability to perform simultaneous bi-directional data transfers between memories (host and device), while executing computations related to one or more kernels. It reduces the GPU's idle time by optimizing the scheduling/issuing of operations associated to distinct kernel executions. However, introducing this optimization in the host's orchestration adds a considerable amount of design complexity, as well as requiring a good knowledge of OpenCL programming. For these reasons, it proves ideal to hide this complexity inside a skeleton, yet letting the developer tweak its parametrization if need be.

Applying concurrency between distinct OpenCL executions is, in itself, a complex exercise. More so, when the mechanism must be general enough to be encapsulated in a skeleton. The scheduling mechanism must optimize the device operation flow. Thus, as the number of skeleton execution requests rises, the framework must be aware, among others: the state of each request (what computation is being done); the resources allocated to the execution, and; how to further advance each execution. However, this by itself does not ensure parallelism. The fine-grain operations (e.g., reads, writes, executions) have to be issued to the OpenCL runtime in a manner that allows their parallel execution. It is not enough simply to launch the operations and expect OpenCL to schedule them in the best way possible. These previous issues gain complexity when the skeleton design includes nesting. Not only must the skeletons support combination between distinct entities, but also, these entities must work together

to introduce concurrency to the execution requests. Consequently, every single skeleton must, at least, be supportive of concurrent executions, even if it does not, by itself, provide the overlap optimization.

Finally, the effectiveness of the overlap is directly proportional to where it is applied on the composition tree. The higher it is, the more sub-trees it affects. Hence, in order to maximize performance, it is always applied by the root node.

Implementation: Supporting multiple concurrent device executions implies the coexistence of multiple datasets in device memory. Therefore, a skeleton must allocate a number of memory objects that enables it to issue operations associated to distinct datasets, in an concurrent and independent manner. This strategy is designated as *multiple buffering*. Consider a skeleton s, as well as a kernel k that is parametrized with one buffer as input and another as output. The configuration of s that uses k to concurrently process three datasets at any given moment, requires the allocation of three sets of memory objects, totalling six memory objects.

The issuing of OpenCL operations to the device is performed via command-queues that offer two execution modes: in-order and out-of-order. The latter schedules the operations according to the device's availability, enabling their parallel execution, as our runtime requires. However, we have ascertained that not every OpenCL implementation supports such queues. Therefore, we opt to build our solution on top of in-order queues, one per set of memory objects. The scheduling responsibility is thus transferred to the Marrow runtime, which must enqueue the operations in such a way that they can be overlapped. This scheme can be scaled out as many times as needed, provided that the platform can supply the resources.

3.4 Supported Skeletons

Marrow currently supports the following set of task and data-parallel skeletons:

Pipeline efficiently combines a series of data-dependant serializable tasks, where parallelism is achieved by computing different stages simultaneously on different inputs in an assembly-line like manner. Considering the significant overhead introduced by memory transfers between host and device memories, this skeleton is ideal for GPU execution since the intermediate data does not need to be transferred back to the host in order to become available to next stage. This execution pattern is suitable for an execution that starts with pre-initialized device memory objects, and is fully able to compute in a shared execution environment. Accordingly, `Pipeline` supports nesting.

Loop applies an iterative computation to a given dataset. The result of each iteration is passed as input to the following (without unnecessary data transfers to host memory), until the final iteration is completed and its results provided as output. This construct supports two computational strategies: one where the loop's condition is affected by data external to the execution domain (a `for` loop), and another where the condition is affected by partial results of every iteration (a `while` loop). Analogously to the `Pipeline`, `Loop` fully supports nesting.

Stream defines a computational structure that confers the impression of persistence of the GPU computation. It achieves this by introducing parallelism between device executions associated to distinct datasets by applying the overlap technique (Subsection 3.3). To simplify the overall framework design only `Stream` provides such functionality. If this behaviour is desirable elsewhere, it is obtainable via nesting on a `Stream`, or its direct usage. Given that applying overlap requires direct control over the input data `Stream` is only qualified as a root node, and thus, is not nestable.

Map (and **MapReduce**) apply a computation to every independent partition of a given dataset, followed by an optional reduction of the results. Considering that this construct is designed for GPU computing, its behaviour differs from the general definition of a *map-reduce* skeleton. Firstly, a GPU should be used to process a large amount of data-elements, so as to compensate for its utilization overheads. Therefore, the input dataset is split into partitions, instead of singular data-elements, being the nested composition tree applied dependently to each of them. Secondly, GPUs are not particularly efficient when reducing a full dataset into a single scalar value. Instead, it is preferable to reduce part of the data in the GPU and return N elements to be finally reduced on the CPU, where N is a power of two larger than a certain threshold (that differs between devices). Thereupon, by default, this construct performs a host-side reduction. Nonetheless, it supports the provision of a reduction kernel, which is applied until the previously cited threshold is reached. Overlap may be efficiently applied between the multiple partition executions, yet, since these skeletons requires direct control over both input and output data, they do not support nesting. Therefore, they offer this feature by resorting to `Stream` internally.

3.5 Programming Example

Marrow's programming model comprises three major stages: *skeleton initialization*, *prompting of skeleton executions*, and *skeleton deallocation*. The first stage holds the `KernelWrapper`'s instantiation and appropriate parametrization, for their subsequent utilization as input parameters to the skeleton instantiation process. This may include nesting if desired by the programmer. In turn, the second stage defines how the application issues execution requests. Given the asynchronism of Marrow's execution model the application may adapt its behaviour to accommodate computations that are executed in parallel to the skeleton requests. Finally, the final stage is trivial. It simply requires the deallocation of the root node, since the latter manages all other skeleton related resources (e.g., inner skeletons, `KernelWrappers`).

Listing 1 illustrates an example that builds a three-staged image filter pipeline fed by a stream. Due to space restrictions we confine the presentation the declaration of a single kernelwrapper (line 2), to the nesting application (lines 3 to 7) and the prompting of the execution requests (lines 8 to 13). A `KernelWrapper` receives as input, the source file, the name of the kernel function, info about the input and output parameters, and the work size. `Pipeline` $p1$ is instantiated with the first two `KernelWrappers` - representing the first two stages. Then, `Pipeline`

$p2$ is created and parametrized with $p1$ along with the last `KernelWrapper`. Ultimately, the `Stream` s is instantiated with $p2$. This scheme creates a composition tree represented by $s(p2(p1))$, in which the kernels associated with the innermost skeleton are computed first. As shown, Marrow's skeletons do not distinguish kernels from nestable skeletons, thus standardizing the nesting mechanism. The prompting of a execution request (line 12) requires the specification of the input and output parameters, and returns a `future` object. In this particular application, the image is divided into segments and discretely feed to the `Stream`.

```
1  // ... instantiate kernel wrappers
2  unique_ptr<IExecutable> gaussKernel (new KernelWrapper(gaussNoiseSourceFile,
       gaussNoiseKernelFunction, inputDataInfo, outputDataInfo, workSize));
3  // instantiate inner skeletons
4  unique_ptr<IExecutable> p1 (new Pipeline(gaussKernel, solariseKernel));
5  unique_ptr<IExecutable> p2 (new Pipeline(p1, mirrorKernel));
6  // instantiate root skeleton
7  Stream *s = new Stream(p2, 3); // Overlap with 3 concurrent executions
8  // request skeleton executions
9  for(int i = 0; i < numberOfSegments; i++){
10   inputValues[0] = ... ; // offset in the input image
11   outputValues[0] = ... ; // offset in the output image
12   futures[i] = s->write(inputValues,outputValues);
13 }
14 //  wait for results ; delete s and resources (e.g. the futures)
```

Listing 1. Stream upon an image filter pipeline

4 Evaluation

The purpose of this study is to measure the overhead imposed by the Marrow framework relatively to straight OpenCL orchestrations, and the impact of the overlap optimization on overall performance. For that purpose we implemented four case-studies in OpenCL (without introducing overlap) and Marrow. All measurements were performed on a computer equipped with a Intel Xeon E5506 quad-core processor at 2.13GHz, 12GB of RAM, and a NVIDIA Tesla C2050 with 3 GB VRAM. The operating system is Linux (kernel 2.6.32-41) linked with the NVIDIA CUDA Developer Driver (295.41).

The first case-study applies a Gaussian Noise filter to an image that is split into non-superimposing segments. The OpenCL implementation processes these segments sequentially, whist the Marrow version submits then asynchronously for concurrent processing. Logically, the latter version adopts the `Stream` skeleton.

The second case-study is a pipelined application of image filters, namely Gaussian Noise, Solarise, and Mirror. Once again, we selected filters that are applicable to non-overlapping segments of an image so as to support an overlapped execution. Consequently, the application performs equivalently to the preceding case-study, differing only by applying multiple filters in succession to each slice. Naturally, the Marrow application uses `Pipelines` nested in a `Stream`.

The third case-study applies a tomographic image processing method, denominated as *Hysteresis*, that iteratively eliminates gray voxels from a tomographic image, by determining if they should be altered into white or black ones. The

Table 1. OpenCL case-studies execution times in milliseconds

	Gaussian Noise (pixels)			Filter Pipeline (pixels)			Hysteresis (MBytes)			N-Body (particles)		
Input parameter size	1024^2	2048^2	4096^2	1024^2	2048^2	4096^2	1	8	60	1024	2048	4096
Execution Time (ms)	3.18	11.82	46.36	3.34	12.46	48.95	402.98	2952.98	19742.80	37.77	78.23	174.61

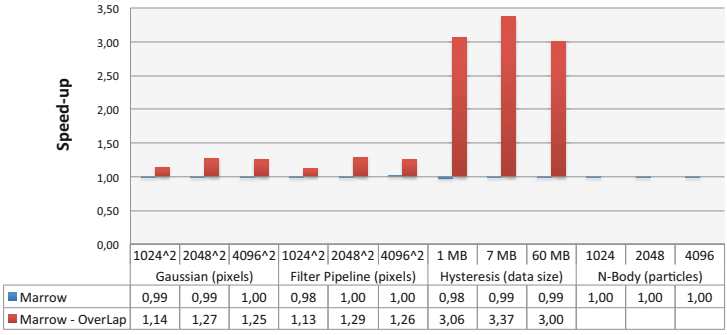

Fig. 2. Speed-up versus OpenCL

algorithm comprises three data-dependent stages, each building upon the results of its predecessor to refine the elimination process. Once more the source images are split into segments, to which the processing performed by each stage is applied independently. However, the size and number of these segments are stage related, and may differ between stages. In any case, at every given stage the computations are iteratively applied to a single segment until the results stabilize. In the OpenCL version, segment processing within each stage is performed sequentially, whist the Marrow version nests a `Loop` into a `Stream` to perform the computation concurrently. Note that the mismatch between corresponding stage related segments prevented us to assemble the `Loop`s in a pipelined execution.

The final case-study is an implementation of the particle-particle N-Body simulation algorithm ($O(n^2)$). In contrast with the previous case-studies the algorithm is not compatible with the partitioning of the input dataset. For that reason, the Marrow version resorts to a single `for` variant of the `Loop` skeleton, that makes no use of the overlap facility.

Performance Results: Table 1 presents the execution times of the OpenCL versions. These measurements isolate the time actually spent on the orchestration and execution of the GPU computation, on an input data of a certain grain, excluding, thus, the initialization and deallocation stages. The depicted values reflect the best results obtained by varying the global and local work size configuration parameters. In turn, Figure 2 displays the speed-up obtained by Marrow relatively to the OpenCL baseline, once more for the best work size configuration. The first version, Marrow, does not introduce overlap, and hence assesses

Table 2. Productivity comparison between distinct versions

	Gaussian Noise	Filter Pipeline	Hysteresis	N-Body
OpenCL basic/with overlap	61/261	81/281	165/365	98/298
Marrow	50	59	222	79

the framework's overhead, while the second, Marrow - Overlap, presents the best result when tweaking the overlap parameter.

The overhead introduced by the framework is minimum, peeking at 2%. Regarding the performance gains brought by the overlap optimization, the first two case-studies show a very similar behaviour. The balance between computation and communication is optimal for the application of our optimization at the medium grain. The Hysteresis' execution pattern differs from the remainder. Its execution flow dictates that after each loop iteration, which processes a single segment, the `Loop` reads the results to host memory, and subsequently evaluates them to access its continuity, a process of complexity $O(N)$ where N is the size of a segment. These two processes are computationally heavy and leave the GPU available to execute upon other datasets. Consequently, we assert the existence of a considerable amount of unexplored parallelism between segment executions. On top of that, the speed-up is incremental given that all three stages introduce it. Hence, it is directly propositional to the amount of overlap, consistently increasing with the latter, up to the maximum number of segments per stage.

Programming Model Evaluation: It comes as no surprise that our programming model is simpler, and of higher-level than OpenCL's, since it orchestrates the whole execution. To somehow quantify this judgement, Table 2 presents the number of lines of code for OpenCL, with and without introducing overlap, and Marrow. To introduce overlap in an OpenCL application we estimated a minimum increase of two-hundred lines of code, adding to the design complexity which would surely grow substantially.

Marrow's programming model productivity trumps the *with overlap* OpenCL versions and consistently requires less code per application than the basic ones. The Hysteresis case-study is an exception, requiring roughly more 40% of code than the OpenCL version. This increase in program size comes as a result of: a) the initializion of three `Loops` nested into three `Streams` is somewhat verbose, and b) the use of the `Loop` skeleton requires the derivation of a base class. Joining these two factors adds a considerable amount of lines of code to the application, justifying the discrepancy between OpenCL and Marrow versions.

5 Conclusions

This paper presented Marrow, a ASkF for the orchestration of OpenCL computations. Marrow distinguishes itself from the existing proposals by: (i) enriching the set of skeletons currently available on the GPGPU field; (ii) supporting skeleton nesting, and (iii) empowering the programmer to easily and efficiently

exploit the ability of modern GPUs to overlap, in time, the execution of one or more kernels with data transfers between host and device memory.

Compared to the state of the art in ASkFs for GPU computing, Marrow takes a different approach. It focuses on the orchestration of complex OpenCL applications rather than the high level expressiveness of simple data-parallel kernels. This allows for a more flexible and powerful framework, whose kernels are bound only by OpenCL's restrictions, and whose skeleton set is richer and more modular. Naturally, as the programmer must express the parallel (kernels) in OpenCL, Marrow's abstraction of the underlying computing model is less effective than the one offered by the remainder.

The accomplished evaluation attested the effectiveness of these proposals. Compared to hand-tuned OpenCL applications that do not introduce overlap, the `Stream` skeleton consistently boosted performance without compromising the simplicity of the Marrow programming model. In addition, the remainder skeletons supply a set of high-level constructs to develop complex OpenCL based applications with negligible performance penalties.

References

1. Udupa, A., Govindarajan, R., Thazhuthaveetil, M.J.: Software pipelined execution of stream programs on GPUs. In: CGO 2009, pp. 200–209. IEEE Computer Society (2009)
2. Dubach, C., Cheng, P., Rabbah, R.M., Bacon, D.F., Fink, S.J.: Compiling a high-level language for GPUs: (via language support for architectures and compilers). In: PLDI 2012, pp. 1–12. ACM (2012)
3. hgpu.org: High performance computing on graphics processing units - Applications, `http://hgpu.org/?cat=11` (last visited in May 2013)
4. NVIDIA Corporation: NVIDIA CUDA, `http://www.nvidia.com/object/cuda_home_new.html` (last visited in May 2013)
5. Munshi, A., et al.: The OpenCL Specification. Khronos OpenCL Working Group (2009)
6. OpenACC: The OpenACC application programming interface (version 1.0) (2011), `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`
7. Sidelnik, A., Maleki, S., Chamberlain, B.L., Garzarán, M.J., Padua, D.A.: Performance portability with the Chapel language. In: IPDPS 2012, pp. 582–594. IEEE Computer Society (2012)
8. Codeplay Software: Offload compiler, `http://www.codeplay.com/compilers/` (last visited in May 2013)
9. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: HLPP 2010, pp. 5–14. ACM (2010)
10. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - a portable skeleton library for high-level GPU programming. In: IPDPS 2011 - Workshops, pp. 1176–1182. IEEE (2011)
11. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. IJHPCN 7(2), 129–138 (2012)
12. AMD Corporation: Bolt C++ Template Library, `http://developer.amd.com/tools/heterogeneous-computing/` (last visited in May 2013)
13. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)