Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs, T. Kuhlen, and F. Marton (Editors)

HyperFlow: A Heterogeneous Dataflow Architecture

Huy T. Vo[1], Daniel K. Osmari[1], João Comba[2], Peter Lindstrom[3] and Cláudio T. Silva[1]

[1]Polytechnic Institute of New York University
[2]Instituto de Informática, UFRGS, Brazil
[3]Lawrence Livermore National Laboratories

Abstract

We propose a dataflow architecture, called HyperFlow, *that offers a supporting infrastructure that creates an abstraction layer over computation resources and naturally exposes heterogeneous computation to dataflow processing. In order to show the efficiency of our system as well as testing it, we have included a set of synthetic and real-case applications. First, we designed a general suite of micro-benchmarks that captures main parallel pipeline structures and allows evaluation of HyperFlow under different stress conditions. Finally, we demonstrate the potential of our system with relevant applications in visualization. Implementations in HyperFlow are shown to have greater performance than actual hand-tuning codes, yet still providing high scalability on different platforms.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

The popularization of commodity multi-core CPUs and multi-GPU units has opened many alternatives for the design and implementation of efficient visualization and data analysis algorithms. However, manually distributing the processing load among CPU cores and GPU units can be very cumbersome. Combining CPU and GPU processing in a more automatic fashion raises several challenges, since CPUs and GPUs behave differently, and require an underlying infrastructure to control execution and data transfer between modules efficiently. Existing pipeline models used in scientific applications, such as systems based on VTK [Kit, BCC*05], already decompose computing tasks into independent modules and transparently manage data communication inside a pipeline. However, these systems were designed around the assumption of a homogeneous processing model, based only on multi-core CPUs. Therefore, a full revision of data structures and algorithms is necessary to leverage the processing power offered by GPUs.

In this paper we introduce HyperFlow, a dataflow architecture that treats all *processing elements* in a heterogeneous computational system as first rate computational units, including multi-core CPUs, GPU units and potentially other types of processing elements. The architecture is designed to run on a single system, which we define as an array of possibly many interconnected nodes. Each node is an individual machine that contains a set of heterogeneous processing elements. The architecture is designed to make full use of data streaming, a concept used before in scientific dataflow systems. HyperFlow provides an infrastructure of abstract layers that allow dataflow systems to be designed without knowledge of the actual processing elements used to execute pipeline modules. Two abstract layers are introduced to separate the module execution context from the actual computation. The first abstraction separate implementations designed for different types of processing elements at the module level. At the processing element level a second abstraction allows execution to be independent of the actual execution context (CPU or GPU). The HyperFlow architecture handles the mapping and coordination between processing elements and execution units. The scheduling is controlled by a dedicated module that relies on a set of strategies to best perform execution tasks.

We first validated HyperFlow with synthetic applications based on a general suite of micro-benchmarks, which were useful for debugging the system and stress testing HyperFlow in common parallel constructions. Tests were also performed with real applications, such as image processing pipelines and isosurfacing. The main contributions introduced in this work are as follows:

DOI: 10.2312/EGPGV/EGPGV12/001-010

- A definition of abstraction layers that can encapsulate pipeline modules and processing elements in different configurations of heterogeneous systems;
- A full infrastructure to enable the construction of modules that can be run on different types of processing elements with full support for data streaming between them;
- Scheduling policies for automatic load balancing tasks across different processing elements;
- Parallel implementation of three different applications: edge detection, streaming multigrid for gradient-domain operations, and isocontouring of structured grids.

2. Related Work

There is a vast literature that discuss parallel programming in heterogeneous systems. Good introductory references is the recent survey on heterogeneous computing of Broddtkorb et al. [BDH*10] and the book by Rauber and Rünger [RR10].

Streaming computation has been a key concept to improve computational performance [KDK*01], essentially due to the increase of parallelism while also reducing memory footprint. The programming model offered by StreamIt [TKA02] and their corresponding systems [DLD*03, CGT*05, TKM*02] created abstractions that used streaming computation for high performance computation. An implementation of the graphics pipeline using StreamIt is given in [CGT*05], with code mapped to the general-purpose Raw architecture [TKM*02]. The need to leverage the parallel processing of GPUs for general-purpose computation led to several programming languages and environments, such as Brook [BFH*04], Scout [MIA*04], CUDA [NVI10], OpenCL [Kro10], Sequoia [FKH*06], etc. These languages expose the GPU as a streaming architecture with parallel programmable units composed of kernels (programs) that operate on streams (data). However, they are considered low-level and they do not support dynamic scheduling across multiple devices.

There are several frameworks proposed to handle distributed computation across heterogeneous many-core systems. XKAAPI [DGLM], Harmony [DY08] and StarPU [ATNW11] proposes runtime environment to allow dynamic scheduling of execution kernels to heterogenous systems resources. During execution, they map the program task-graph to available resources using a specific data plan. The relationship between data and tasks specification here are coupled much loosely than that of an actual dataflow architecture. A parallel dataflow framework was introduced in [VOS*10] for many-core systems. Their proposed architecture distributes computing resources (e.g. threads) to modules per execution request. Modules have only one implementation and can only run on a fixed set of resources.

Support for heterogeneous systems composed of hybrid CPU/GPU systems are starting to appear recently. Instead of forcing applications to be rewritten into a streaming processing format, HMPP [BB09] enhances CUDA programs with a set of compiler directives, tools and software runtime that support multi-core CPU parallel programming. FastFlow [ATM09] is a low-level programming framework that support streaming applications for multi-core machines at fine-grained computations. The CUDASA programming language [MFS*09] extends CUDA to allow distributed computation on multiple GPUs in a local system or across machines on a network. GRAMPS [SFB*09] is a programming model specifically designed for graphics pipelines that allows task and data-level parallelism for multi-core systems, composed of either CPUs or GPUs, but not both.

Anthill [THCF10] uses a demand-driven approach based on a performance estimator to assign tasks to processors in CPU/GPU systems. The DAX [MMGA] projects under way are also addressing CPU/GPU parallel frameworks for dataflow systems. In [WS10], an approach called Elastic Computing that separates functionality from implementation was proposed. In this work we propose similar abstractions, described in more details in the following section.

3. HyperFlow Abstractions

The design of HyperFlow relies on the construction of abstraction layers. At the module level, pipelines are defined as interconnected *Task-Oriented Modules (TOMs)*, and executed as a set of token-based data instances called *flows*. To encapsulate available processing units, a Virtual Processing Element (VPE) forms an abstraction layer over the actual computing resources available in the system.

3.1. Task-Oriented Modules (TOMs)

A pipeline in HyperFlow consists of a set of interconnected Task-Oriented Modules (TOMs). Each TOM defines and holds parameters needed for a specific computational task, such as the number of input and output ports. To allow the same pipeline to be executed across a wide range of different computational resources transparently, TOMs do not explicitly represent task implementations. Instead, they store a list of *task implementation* objects, which are dynamically scheduled to perform the actual computation on a given set of inputs. This separation of task specification and implementation is one of the main differences between HyperFlow and similar systems. The requirement for a TOM to be executed at runtime is for it to have a task implementation that matches the system resources (e.g. CPUs or GPUs).

3.2. Flows

Similar to token-based hardware dataflow architectures, HyperFlow executes pipelines by sending instruction tokens to processing units for execution, and sending data tokens back as results. Both of these token are modeled as a *flow* in HyperFlow, which passes between connected TOMs in the pipeline. Each flow contains a data reference along with

its meta information, such as source and destination modules, to control pipeline executions as well as streaming data across modules. Flows are classified as *waiting, live, or dead* depending on their status (waiting for execution, executing, and finished, respectively). After a module completes execution, new flows might be generated for subsequent modules in the pipeline to process.

Pipeline execution in HyperFlow execution does not restrict pipelines to be directed acyclic graphs (DAGs). While other systems make use of DAGs to guarantee the module execution order, it comes at a cost of limited pipeline parallelism and static execution. HyperFlow, on the other hand, supports dynamic execution for enabling pipelines with feedback communication. In order to determine when a module is ready to execute, HyperFlow maintains a flow cache that stores incoming flows until all of the input data arrives, and then trigger the module execution.

3.3. Virtual Processing Elements (VPEs)

One main feature of HyperFlow is the ability to schedule pipeline modules for execution across heterogeneous computing resources such as GPUs and CPUs in a transparent way. To manage different computational resources, we introduce the concept of *Virtual Processing Elements (VPEs)*, which are abstract layers that manage execution contexts of specific computing resources. VPEs are designed as a service that waits for tasks to be executed when a resource becomes available. Once this happens, a VPE first verifies if all input data is properly transferred to its current context, since data may reside in different, mutually inaccessible, memory areas, such as CPU and GPU memory. In HyperFlow, we assume that each VPE, regardless of their underlying hardware, has access to main CPU memory. Therefore, a data-transfer path between potentially very different VPEs is always possible, although users are free to implement their own data transfer routines.

4. HyperFlow Architecture

In this section we describe the HyperFlow architecture for parallel execution of dataflows on shared-memory systems (Fig. 1). By design, HyperFlow allows a single module to have more than one implementations and depending on the system it runs on, the most appropriate one(s) will be executed. At the application developer level, the framework provides a C++ template API that dynamically allows construction and execution of pipelines.

4.1. Execution Engine (EE)

The main controlling component of HyperFlow is the *Execution Engine* (EE). At runtime, the EE initializes a set of VPEs that are mapped to the corresponding computing resources available and managing all flows passing through

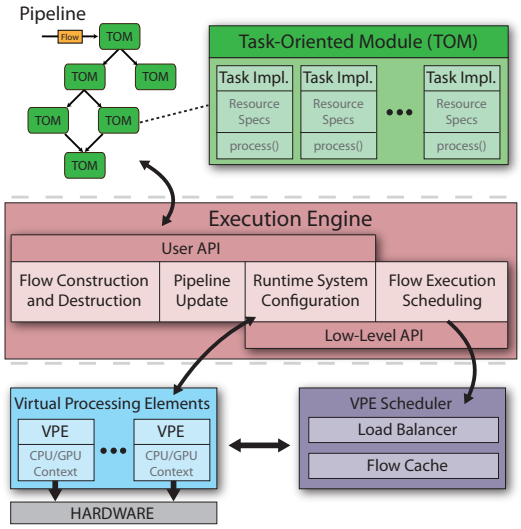


Figure 1: *HyperFlow Architecture. The execution engine is the control module, managing live flows generated by TOMs. The scheduler assigns flows for execution in available VPEs.*

the system, as well as the status of VPEs. A polling thread is the main control module of the EE, which waits in a non-blocking way for either a waiting flow to be generated in the TOMs, or a resource VPE to become available. Once a waiting flow is generated, the EE dispatches it to the VPE scheduler for execution. This polling thread also instructs the VPE scheduler when a VPE resource becomes available.

4.2. VPE Scheduler

The VPE scheduler is responsible for managing and scheduling flows for execution on available VPEs. Internally, it maintains two priority queues of flows: a *waiting queue* which contains flows generated by the EE, and a *live queue* of flows currently executing. Flows sent from the EE are initially added to the VPE waiting queue and sorted by their identification number. HyperFlow generates these numbers monotonically and uses them to quickly determine the execution order of pipeline modules such that global scheduling strategies can be employed without actually traversing the whole pipeline.

Flow identification also allows the scheduler to cache input flows for detecting when a module is ready for execution. In a streaming pipeline where data from different timesteps can simultaneously flow through a single module, the cache is necessary to make sure that module inputs are sorted out appropriately before getting processed. Only when all input flows with the same identification number, or the same time step, are present in a module cache, the module will be scheduled for execution with that set of flows.

The scheduler in HyperFlow is designed using the event-

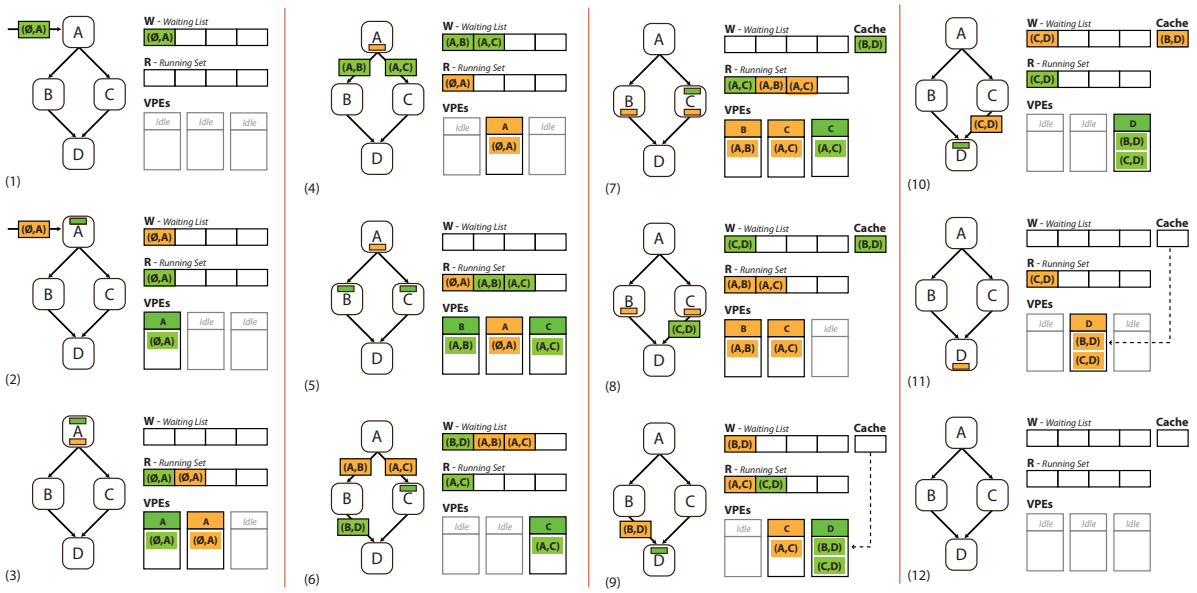


Figure 2: *Processing two data blocks using HyperFlow. A flow between modules A and B is identified by* (A,B)*, and is colored green for the first data block and orange for the second. (1) the first (green) flow* $(\emptyset,A)$ *is generated and goes to the waiting queue. (2) then it moves to the live queue and triggers execution of module green A in the VPE. (3) it is executed in the VPE. (4) flow A terminates, and generates* (A,B) *and* (A,C) *flows. (5) and trigger execution of B and C modules. (6) B terminates and generates* (B,D)*. (7)* (B,D) *moves to the cache since D can only execute when* (C,D) *is live. (8) C terminates and generates* (C,D)*. (9)* (C,D) *becomes live, and along with* (B,D) *in the cache trigger D to execute. (10) D is in execution. (11) D terminates. The same process is repeated for the second data block starting with the orange flow* $(\emptyset,A)$ *in step (2) to (12).*

driven paradigm but allowing developers to provide their own scheduling strategy. The default scheduling strategy in HyperFlow is based on time statistics. As HyperFlow starts executing a pipeline, it attempts to execute each module on at least a CPU VPE and every GPU VPE to establish an initial average time. For subsequent executions of the module, the average time of past executions on each VPE will be used to select an available VPE with the fastest average time.

The default scheduling strategy also allows interruptions when new scheduling events are posted to the scheduler. This is to ensure the scheduler to favor depth-first executions, pushing data as far as possible downstream. The main advantage of this approach is the cache-coherent access pattern on hierarchical memory architectures of both CPUs and GPUs. It also reduces the overall memory footprint, thus, increasing the amount of in-core/on-device data that can be processed concurrently. Since the scheduler only operates on a copy of the waiting queue in order allow simultaneously scheduling and updating the waiting queue, when a VPE is done executing a flow and generating new flows, it might be desirable to utilize that recently available VPE to execute those newly created flows as well. Therefore, a restart of the scheduling is required to facilitate depth-first execution.

It is also necessary to note that in the case when a data transfer across VPEs is needed, the scheduler will first tell the source VPE to convert the flow data to the main CPU host memory. The VPE has to be responsible for this conversion instead of the VPE scheduler because it is not always possible for the VPE scheduler to access the internal memory context of each flow due to driver regulation. After the conversion is completed, newly converted flows will be passed on to the destination VPE for further processing. The fact that HyperFlow automatically takes care of this conversion process in a thread-safe manner allows GPU code to not worry about thread migration issues. For instance, in CUDA, GPU memory that are allocated inside a thread can only be used or freed on that same thread. Thus, in order to perform computation across two GPUs, users have to normally go into the first GPU thread, copy the data to main memory, then in the second thread, another copy has to be performed to transfer the data to the second GPU thread. Since HyperFlow always perform data transfer in this fashion, this is no longer a problem.

4.3. Classes of Parallelism Supported

HyperFlow supports a combination of task, data and pipeline parallelism. Task and data parallelism are natively supported in HyperFlow using the default scheduling strategy. As long

as there are available VPEs, the VPE scheduler will concurrently assign *independent flows*, i.e. flows with different ids or without any upstream/downstream relations, for execution. Since independent tasks and distinct input data blocks produce independent flows, HyperFlow can interchangeably coordinate between task and data parallelism. Because flows with smaller ids have higher priority on the waiting queue, HyperFlow favors task parallelism over data parallelism. This allows HyperFlow to maintain a small memory footprint with better cache behavior, as an individual data block travels as far down the pipeline as possible before new data blocks are processed. Since HyperFlow automatically instantiates modules as part of the data-parallelism paradigm to utilize all computational resources, each module may have more than one instance at any time, thus not strictly enforcing pipeline parallelism. In this case, pipeline parallelism is rather promoted to "streaming" data parallelism, where threads are only required to execute different data-blocks on a sub-network. However, if a group of task implementations is not reentrant, i.e. cannot be run with multiple instances, data will be passed through that sub-network with true pipeline parallelism. Currently, HyperFlow allows users to specify a task implementation to be not reentrant by using an internal mutex locking mechanism. An example of a 12-step pipeline running with various types of parallelism is shown in Fig. 2. Task and data parallelism are in effect at steps (5,7) and (3,7), respectively. Pipeline parallelism occurs at steps (5,9), and the flow cache is used in (7-11).

4.4. Memory Management

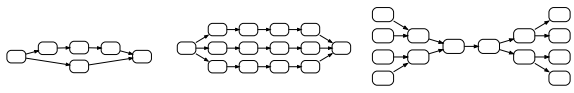
HyperFlow supports execution on different devices (such as CPUs and GPUs, for instance) interchangeably, which requires the transfer of data objects between devices. For this purpose, we enforce that all data objects must inherit from the predefined `Data` class. This class implements reference-counted objects that are required to be copied on write This approach has two advantages: one is the obvious smaller memory footprint, since multiple references to the same object use much less space than copies of this object. Also, the copy-on-write approach implies that HyperFlow has no read-after-write or write-after-write hazards, typical of many concurrent programming approaches. However, for a pipeline that passes a large amount of data, data copy can be very expensive. In this case, users may override the `Data` class to use a different approach.

A data object in HyperFlow must also define its *medium* and *conversion procedures*. Currently there are two types of data mediums in HyperFlow, `DM_CPU_MEMORY` and `DM_GPU_MEMORY`, which specify the main memory and GPU device memory, respectively. The purpose of the conversion procedure, `createInstance(DMTYPE medium)`, is to allow the engine to request a new data object with a specific type of medium that is equivalent to the original data. This allows data to be passed between different VPEs in a customizable manner. As was discussed previously, we require that all `Data` objects implement at least a conversion to and from main CPU memory, to ensure that there will always be at least one data-transfer path between any pair of VPEs. Similar to data creation, data deletion must run on the original context in which the `Data` object was created. Therefore, when a `Data` object reaches zero references, the EE schedules the VPE to release its data resources, instead of doing so directly.

Given the spontaneous mapping of VPEs and module implementations in HyperFlow, dynamic memory allocation and deallocation are unavoidable. While these operations are acceptable on CPUs, they are considerably slow on GPUs. For example, thousands calls to `cudaMalloc()` and `cudaFree()` could take seconds to complete. HyperFlow addresses this issue by providing a customizable memory allocator for each VPE. Pipeline developers may use this to assign an optimal memory plan for their applcation. By default, each GPU VPE in HyperFlow has a fixed-size block allocator where its total size and block-size can be changed programmatically. If an application has to allocate many objects with a similar size, they can take advantage of `localAlloc()` and `localFree()` of the GPU VPE to only call `cudaMalloc()` only once. An example demonstrating data management of HyperFlow is given in the supplemental material.

Along with the depth-first scheduling strategy that at the same time, aiming to retain data on the same VPE, data streaming is fully supported in HyperFlow by having flows "streaming" data from one module to another. The combination of these designs in streaming allows HyperFlow to achieve high performance by hiding latency with coherency of memory accesses.

5. Synthetic Applications



(a) asymmetric (b) split-join (c) scan

Figure 3: *Topology of micro-benchmarks.*

In this section we focus in a set of micro-benchmarks designed to stress test both computational performance and data bandwidth. Experiments were conducted on two systems: (A) an i7 (8 HT cores), 6 GB of RAM; 3 GPUs: 2x GeForce GTX 295 and 1x Tesla C1060 with 4 GB; (B) SGI UV with 96 Xeon cores and 1 TB of RAM.

5.1. Micro-Benchmarks

Micro-benchmarks were designed to evaluate scheduling as well as data handling strategies. The idea is to implement modules that keep the computational device occupied just

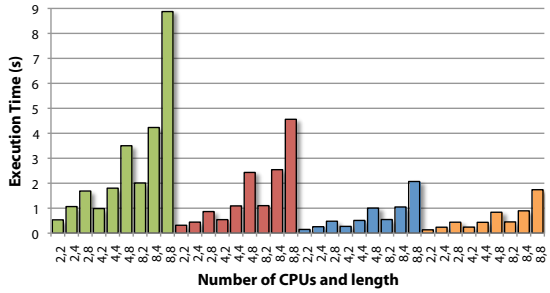


Figure 4: *Split-join benchmarks using 1, 2, 4 and 8 threads (green, red, blue and orange bars respectively) and 1000 flows with width, length* $= 2, 4, 8$.

like a regular implementation but without the need of informing actual code. To allow different pipeline configurations, we designed a framework that generates benchmark programs from a description of basic network topology and module description. Fig. 3 illustrates the main benchmark classes used: asymmetric, split-join, and scan.

The asymmetric benchmark is used to stress unbalanced computation, where data reaches the destination much earlier through one path, causing many flows to be queued up at the final module. It is desirable that flow scheduling in the longer path is not affected by this bottleneck. This benchmark is parametrized by the number of modules on the longest path.

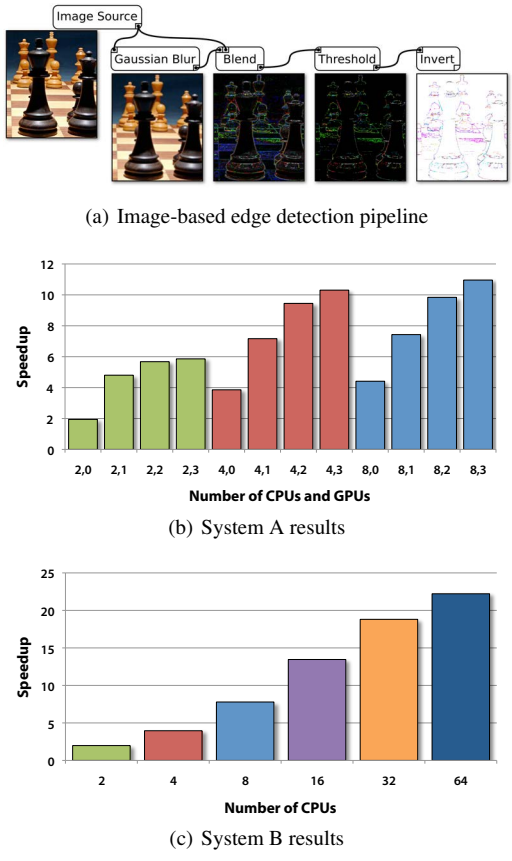
The split-join benchmark model pipelines that divide the input into independent sequences of computation. This benchmark is parametrized by its length and width. This benchmark has the highest degree of parallelism up until the last module, which merges the pieces back together. A linear benchmark that models a typical streaming pipeline can be seen as a subset of this benchmark. The scan benchmark model the typical parallel operation by the same name, using the number of reduction steps as parameter.

5.2. Performance Results

We conducted several experiments using a great variety of micro-benchmarks, and list below the main results obtained. To evaluate the performance of the scheduler, we extracted execution traces containing start and finish times for each module execution. We investigated this data using the animation of flows passing through the pipeline with a Gantt chart indicating when each VPE is active processing a flow. Accompanying video shows the complete animation. results for the split-join benchmarks running with 1000 flows. Runs with 10 and 100 flows as well as with the scan and assymetric benchmarks produced similar graphs.

6. Real-case Applications

In this section, we report comparison results between HyperFlow and hand-tuned implementation of three applications.



(a) Image-based edge detection pipeline

(b) System A results

(c) System B results

Figure 6: *Edge detection pipeline on approximately 4GB of image data, comparing to a system with 1 CPU and 0 GPUs.*

6.1. High Throughput Image Processing

The first application to illustrate HyperFlow is an edge detection framework capable of dealing with multiple images concurrently. The pipeline starts with a source TOM which decodes a stream of images from disk and send them down to another TOM that performs Gaussian blur. The next step in this pipeline consists of a blending operation that returns the difference between the original image and the blurred version. The combined image is streamed to a threshold TOM that computes the image accumulated histogram and discards pixels whose accumulated frequency fall outside a given range (we set this range to be between 95% and 100% of the total accumulated histogram value). Finally, the pipeline sends images to an inversion TOM for display preparation. The main form of parallelism in this pipeline is the streaming data-parallelism.

Each TOM in this pipeline, except the decoding one, has dual implementations, one for CPU and one for GPU. At runtime, depending on the available resources, HyperFlow will instantiate a number of these implementations to handle incoming images on different VPEs concurrently. Fig. 6(a)

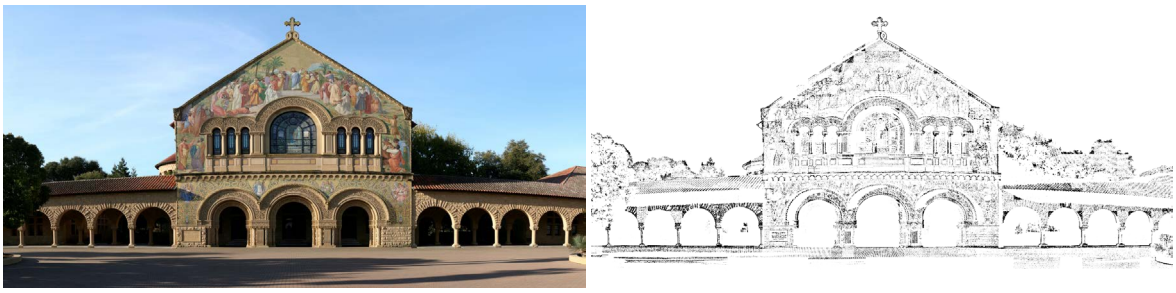


Figure 5: *Edge detection of the Stanford church computed using HyperFlow: (left): 512 input images of 3 Mp each (1.5 Gp total); (right): result computed in* 7 *seconds using a heterogeneous system composed of* 8 *CPU threads and* 2 *GPUs.*

Scheduling Strategy	DFS	Host Memory	Fermi Memory	Non-Fermi Memory
Time Statistics	Yes	2.0 GB	60 MB	60 MB
	No	3.3 GB	216 MB	252 MB
Greedy	Yes	742 MB	48 MB	36 MB
	No	1.3 GB	228 MB	72 MB
Both GPUs CPU Threshold	Yes	1.9 GB	48 MB	48 MB
	No	3.3 GB	156 MB	168 MB
All CPU	Yes	708 MB	-	-
	No	1.5 GB	-	-

Table 1: *Memory usage with different scheduling strategies.*

illustrates this pipeline and how simple it is to integrate an existing code into HyperFlow. It shows the function used for image inversion, as well as a small class that wraps this function to provide an implementation to the image inversion TOM. Fig. 6(b,c) report speedups for both system A and B.

6.1.1. Heterogeneous Scheduling

Though system A is comprised of heterogeneous processing elements (CPUs and GPUs), all GPU devices are indeed homogeneous in term of their compute capability as a CUDA 1.3 device. In other words, a GPU implementation would always produce the same performance regardless of which GPU it is mapped to in this system. Thus, we further experiment HyperFlow on another system to better validate our scheduling strategy on fully heterogeneous platforms. This system consists of 8 Xeon cores @ 3.2GHz, 1x GTX 480 (Fermi, CUDA 2.0), 1x Quadro FX 5800 (Non-Fermi, CUDA 1.3) and 24 GB of RAM.

One of the shortcomings of early CUDA-based GPUs is the slow performance of atomic operations on global memory. This issue has been resolved by the Fermi architecture [NVI11] with the new device L2 cache. As atomic operations are heavily used for computing accumulated histograms in the GPU implementation of the threshold module, a good scheduling strategy should avoid running this implementation on a non-Fermi card for the edge detection pipeline. HyperFlow achieves this by analyzing past execution times of the threshold module.

Fig. 7 reports performances for different scheduling strategies and pipeline configurations in HyperFlow. *All CPU* and *All GPU* refer to pipelines where each module has

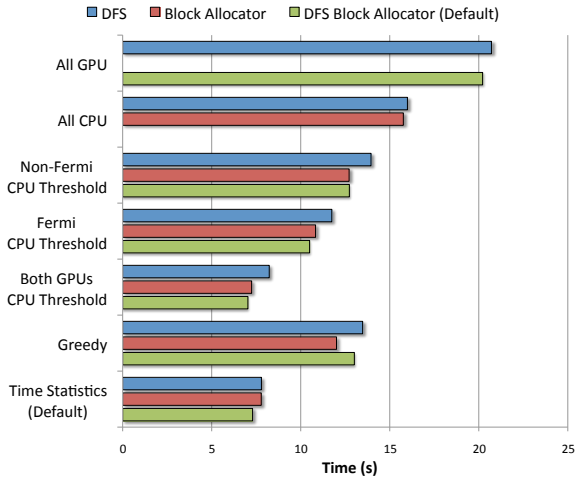


Figure 7: *Performance results for the edge detection of the Stanford dataset with different scheduling strategies.*

only a single implementation that runs on CPU or GPU, respectively. As shown in the figure, forcing the pipeline to run completely on the GPUs results in the worst performance, mostly due to having the non-Fermi card overwork on the threshold TOM. As we restrict the threshold TOM to run only on CPUs while allowing the rest to be freely deployed on the non-Fermi card, the Fermi card or both, the results are much better as shown in the *Non-Fermi-*, *Fermi-*, and *Both GPUs- CPU Threshold* graph, respectively. In the last two entries, we remove this restriction and let HyperFlow pick the best possible scheduling. *Greedy* refers to the strategy of mapping whichever VPE that is available VPE to waiting flows. *Time Statistics* is the default strategy, as described in Section 4.2, that can automatically blacklist the non-Fermi card from running the threshold module as a result of its slow average execution time.

Our experiments also show that the introduction of block allocators to GPU VPEs has actually added a noticeable improvement on the scheduler performance. On the other hand, the depth-first strategy gives similar (if not faster) performance in most cases, but uses considerably much less mem-

ory (see Table 1). There is no *Block Allocator* result reported for *All GPU* in Fig. 7 because without the DFS enforcement, the amount of GPU memory required to run the experiment exceeded the physical limit of 1.5 GB on the GTX 480 card.

6.1.2. Comparison to VTK

Our last validation of this application is the comparison of HyperFlow to the de-facto standard visualization API, VTK, in particular, their multi-threaded imaging library. Fortunately, VTK provides implementations for all of the modules above except for the image decoder which was designed specifically for our application. Fig. 8 shows the performance results of VTK, HyperFlow, and their variations on the system described in the previous section.

The *VTK* line refers the original implementation of VTK, which did not scale well. This is because VTK can only parallelize execution at module level (dividing the pixels) with multithreading. Therefore, as one of the limitations, it could not perform image decoding in parallel. In our experiment, the parallel dataflow framework by [VOS*10] that has been made available through VTK, also produced a similar performance result as the original VTK. This is mainly because such framework only promotes task- and data-parallelism but there is only a single path of execution in the pipeline.

We further analyze the performance of a manually optimized version of the above VTK pipeline (shown as *Hand-tuned VTK*). The pipeline first scan through the input stream for partitioning decoded data. Then, it creates a number of threads where each of them will decode a set of images. While being relatively complex, this manually constructed pipeline is still outperformed by the all CPU implementation in HyperFlow (shown as *HyperFlow CPU*). The pipeline in HyperFlow simply produced less overhead and only need to stream data from disk once. The green line shows the performance of HyperFlow when also utilizing the 2 GPUs, something that VTK currently does not support.

6.2. Streaming Multigrid Gradient-Domain Processing

This experiment is to demonstrate the streaming capabilities of HyperFlow against another hand-tuned implementation. We construct a pipeline similar to the streaming panorama stitching by Kazhdan et. al [KH08] but using HyperFlow with a simple modification to the multigrid solver that allows relaxation steps to be updated in parallel automatically at runtime. Instead of performing temporally blocked relaxation in a single streaming operation, the Gauss-Seidel iterations are separated and chained together (as shown on the left). Hence, pipeline parallelism can be triggered automatically by HyperFlow as rows of images are streamed through this solver. However, since each iteration of the update also has its own streaming buffer, rows needs to be copied among iterations. This results in a higher memory footprint and bandwidth. Thus, there is a trade-off between concurrency and memory usage in this pipeline.

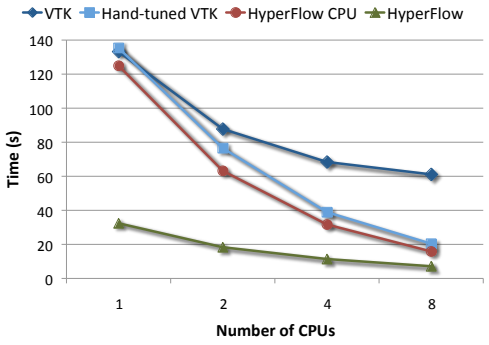


Figure 8: *Performance comparison of HyperFlow vs. VTK.*

Image	Original	HyperFlow	Speedup
PNC3	39.78s	31.68s	1.26x
Edinburgh	73.73s	55.06s	1.34x
Red Rock	126.41s	90.06s	1.40x

Table 2: *Performance results for streaming multigrid gradient-domain image editing in HyperFlow.*

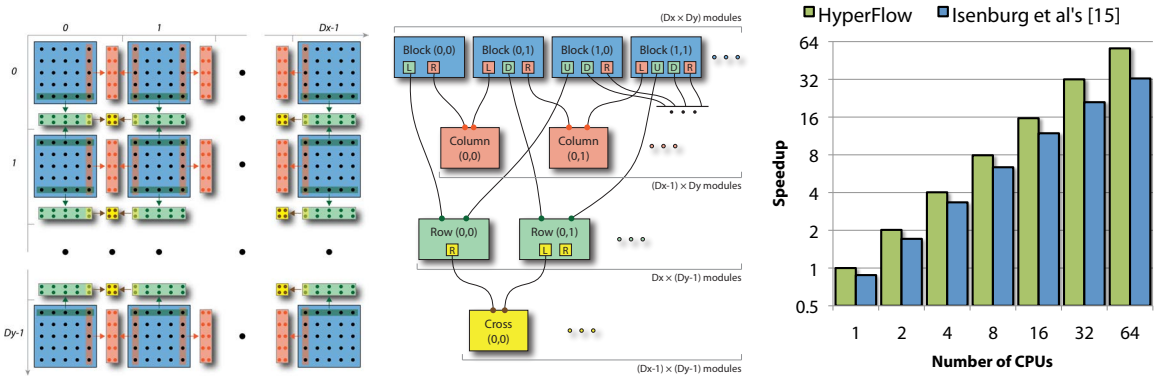
Table 2 shows performance timings for the modified pipeline in HyperFlow. Though the original implementation was very well optimized, it is shown that the performance can still be improved by a simple transformation into HyperFlow. However, the amount of parallelism here depends directly on the number of iterations involved. Due to the small number of iterations needed by the application, i.e. 5, and the additional data copies between iterations, HyperFlow was only able to achieve the maximum of 1.4x speedup.

6.3. Isocontouring Structured Grids

We conclude our evaluation with a more challenging load balancing problem: data-parallel isosurface computation. As input we use the RMI dataset from LLNL [ILC10]: a $2048 \times 2048 \times 1920$ regular grid of single-byte values totaling 8 GB of data. This dataset was partitioned into blocks of size $128 \times 128 \times 64$ voxels, with each mapped to a separate flow. Because some blocks contain no isosurface while others may be quite dense, a static assignment of blocks to processors can introduce considerable load imbalance; something that HyperFlow avoids.

The contouring computation cannot process each block independently, because contours must be extracted also *between* voxels residing in adjacent blocks. Fig. 9(a) illustrates (using a 2D analogy) the data communication needed between adjacent blocks, here shown in blue. The red and green thin blocks correspond to shared block "edges"; yellow blocks are shared corners. Note in particular that this duplication of block boundary voxels adds a considerable number of thin layers as additional flows processed by HyperFlow. These flows and data dependencies are further illustrated by the pipeline in Fig. 9(b).

We compare the performance of HyperFlow with Isen-



(a) data communication (b) pipeline structure (c) performance comparison

Figure 9: *Parallel isosurface extraction in HyperFlow and comparison result to the approach of Isenburg et al's [ILC10].*

burg et al.'s [ILC10] implementation of their parallel streaming isocontouring algorithm—here referred to as "Ghost"—which pads each block with a surrounding layer of *ghost zones* (voxels from adjacent blocks) corresponding to the red, green, and yellow blocks in Fig. 9(a). The Ghost algorithm was designed for distributed computation, and therefore uses MPI for (in-memory) interprocess communication. Fig. 9(c) summarize the execution time for both methods. As is evident, both methods achieve near-linear speedup, with HyperFlow consistently outperforming Ghost. We conjecture that this result is due in part to better load balancing in HyperFlow, which contrary to Ghost performs a dynamic rather than static task allocation, and MPI overhead in Ghost.

7. Discussion

Pipeline developers often spend a considerable amount of time tweaking workflows to make sure they optimally utilize available resources, which requires manually writing modules to run either on the CPU or the GPU. This approach has a significant portability drawback, since the same pipeline, when executed on a different system, might not run with the same efficiency, or not even run at all (if the target machine lacks the necessary computational resources). We addressed this problem with the concept of a TOM, in which tasks may have more than one implementation. This abstraction enforces a clean separation between module *specification* and *implementation*, thus, allowing developers to construct efficient, high-level pipelines without prior knowledge of the computing resources available at execution time. For example, in HyperFlow the exact same pipeline can be executed on machines with different numbers of CPU cores and GPU devices by compiling tasks into Callable Objects, such as CPU function pointers and GPU kernels. The scheduler places these into appropriate execution contexts, such as CPU masks or individual GPU devices. It is important to observe that to support pipeline parallelism, these objects need to be reentrant. In our current implementation, they are simply compiled as function pointers for local execution. To allow HyperFlow to run across a distributed system, they would need to be compiled into executable files that can be sent over the network.

Although the examples discussed in this paper include VPEs defined only for CPU threads and GPU devices, HyperFlow is designed to allow arbitrary VPEs, with no inherent limitation on the kind of computing resource that can be integrated into the architecture. Therefore, it is possible to define VPEs that exploit many other computational resources, such as Web and Cloud Computing, among others.

8. Conclusions and Future Work

We presented HyperFlow, a parallel dataflow architecture that allows pipelines to take full advantage of modern, heterogeneous computational systems. HyperFlow is comprised of several components carefully designed to create an efficient abstraction layer that allows developers to design pipelines without knowing on which types of processors their modules get executed. The Task-Oriented Module encapsulates the difference between task specification and implementation. Therefore, pipeline modules can have multiple heterogeneous implementations, individually designed to take advantage of a particular type of processor. Pipeline execution relies on a token-based mechanism in which flows are sent to processing units. We also introduced the notion of a Virtual Processing Element, an abstraction that allows processing elements to be offered without explicitly knowledge whether they are implemented as a CPU thread or a GPU device. In the future, we plan to extend HyperFlow to distributed systems with heterogeneous configurations, re-visit architectural decisions, and extend validation with applications of even higher computational complexity.

Acknowledgements

We would like to thank Marc Levoy at Stanford University for the David model and Bill Cabot, Andy Cook, and Paul Miller at LLNL for the Rayleigh-Taylor dataset. We also thank Luiz F. Scheidegger for his help on the development of our system. This work was supported in part by the National Science Foundation (CCF-08560, CCF-0702817, CNS-0751152, CNS-1153503, IIS-0844572, IIS-0904631, IIS-0906379, IIS-1153728, and NIH ITKv4), the Department of Energy, CNPq (processes 200498/2010-0, 569239/2008-7, and 491034/2008-3), IBM Faculty Awards and NVIDIA Fellowships. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[ATM09] ALDINUCCI M., TORQUATI M., MENEGHIN M.: *FastFlow: Efficient Parallel Streaming Applications on Multi-core*. Tech. Rep. TR-09-12, UniversitÃ di Pisa, Dipartimento di Informatica, Italy, Sept. 2009.

[ATNW11] AUGONNET C., THIBAULT S., NAMYST R., WACRENIER P.-A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper. 23* (February 2011), 187–198.

[BB09] BODIN F., BIHAN S.: Heterogeneous multicore parallel programming for graphics processing units. *Sci. Program. 17*, 4 (2009), 325–336.

[BCC*05] BAVOIL L., CALLAHAN S., CROSSNO P., FREIRE J., SCHEIDEGGER C., SILVA C., VO H.: VisTrails: Enabling interactive, multiple-view visualizations. pp. 135–142.

[BDH*10] BRODTKORB A. R., DYKEN C., HAGEN T. R., HJELMERVIK J. M., STORAASLI O. O.: State-of-the-art in heterogeneous computing. *Sci. Program. 18* (January 2010), 1–33.

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3 (2004), 777–786.

[CGT*05] CHEN J., GORDON M. I., THIES W., ZWICKER M., PULLI K., DURAND F.: A reconfigurable architecture for load-balanced rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), HWWS '05, ACM, pp. 71–80.

[DGLM] DANJEAN V., GAUTIER T., LAFERRI C., MENTEC F. L.: Xkaapi. http://kaapi.gforge.inria.fr/XKaapi/XKaapi.html.

[DLD*03] DALLY W. J., LABONTE F., DAS A., HANRAHAN P., AHN J.-H., GUMMARAJU J., EREZ M., JAYASENA N., BUCK I., KNIGHT T. J., KAPASI U. J.: Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2003), SC '03, ACM, pp. 35–.

[DY08] DIAMOS G. F., YALAMANCHILI S.: Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing* (New York, NY, USA, 2008), HPDC '08, ACM, pp. 197–200.

[FKH*06] FATAHALIAN K., KNIGHT T. J., HOUSTON M., EREZ M., HORN D. R., LEEM L., PARK J. Y., REN M., AIKEN A., DALLY W. J., HANRAHAN P.: Sequoia: Programming the memory hierarchy. In *SC 2006 Conference, Proceedings of the ACM/IEEE* (2006), p. 4.

[ILC10] ISENBURG M., LINDSTROM P., CHILDS H.: Parallel and streaming generation of ghost data for structured grids. *Computer Graphics and Applications, IEEE 30*, 3 (2010), 32–44.

[KDK*01] KHAILANY B., DALLY W., KAPASI U., MATTSON P., NAMKOONG J., OWENS J., TOWLES B., CHANG A., RIXNER S.: Imagine: media processing with streams. *Micro, IEEE 21*, 2 (2001), 35 –46.

[KH08] KAZHDAN M., HOPPE H.: Streaming multigrid for gradient-domain operations on large images. *ACM Trans. Graph. 27* (August 2008), 21:1–21:10.

[Kit] KITWARE: The Visualization Toolkit (VTK) and Paraview. http://www.kitware.com.

[Kro10] KRONOSGROUP: OpenCL. http://www.khronos.org/opencl/, 2010.

[MFS*09] MULLER C., FREY S., STRENGERT M., DACHSBACHER C., ERTL T.: A compute unified system architecture for graphics clusters incorporating data locality. *IEEE Transactions on Visualization and Computer Graphics 15* (2009), 605–617.

[MIA*04] MCCORMICK P. S., INMAN J., AHRENS J. P., HANSEN C., ROTH G.: Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *Visualization '04* (2004), IEEE, pp. 171–178.

[MMGA] MORELAND K., MA K.-L., GEVECI B., AYACHIT U.: Dax. http://www.daxtoolkit.org/index.php/Main_Page.

[NVI10] NVIDIA: CUDA programming guide. http://developer.nvidia.com/object/cuda.html, 2010.

[NVI11] NVIDIA: Fermi compute whitepaper. http://www.nvidia.com/object/IO_89570.html, 2011.

[RR10] RAUBER T., RÜNGER G.: *Parallel Programming: for Multicore and Cluster Systems*. Spri, 2010.

[SFB*09] SUGERMAN J., FATAHALIAN K., BOULOS S., AKELEY K., HANRAHAN P.: Gramps: A programming model for graphics pipelines. *ACM Trans. Graph. 28* (February 2009), 4:1–4:11.

[THCF10] TEODORO G., HARTLEY T. D. R., CATALYUREK U., FERREIRA R.: Run-time optimizations for replicated dataflows on heterogeneous environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (New York, NY, USA, 2010), HPDC '10, ACM, pp. 13–24.

[TKA02] THIES W., KARCZMAREK M., AMARASINGHE S. P.: Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction* (London, UK, 2002), CC '02, Springer-Verlag, pp. 179–196.

[TKM*02] TAYLOR M. B., KIM J., MILLER J., WENTZLAFF D., GHODRAT F., GREENWALD B., HOFFMAN H., JOHNSON P., LEE J.-W., LEE W., MA A., SARAF A., SENESKI M., SHNIDMAN N., STRUMPEN V., FRANK M., AMARASINGHE S., AGARWAL A.: The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro 22* (March 2002), 25–35.

[VOS*10] VO H. T., OSMARI D. K., SUMMA B., COMBA J. L. D., PASCUCCI V., SILVA C. T.: Streaming-enabled parallel dataflow architecture for multicore systems. *Computer Graphics Forum 29*, 3 (2010).

[WS10] WERNSING J. R., STITT G.: Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. *SIGPLAN Not. 45* (April 2010), 115–124.