2016 IEEE International Parallel and Distributed Processing Symposium Workshops

Heterogeneous Streaming

Chris J. Newburn, Gaurav Bansal, Michael Wood, Luis Crivelli, Judit Planas, Alejandro Duran
chris.newburn@intel.com, gaurav2.bansal@intel.com, michael.wood@3DS.com, luis.crivelli@3DS.com, judit.planas@epfl.ch, alejandro.duran@intel.com
Paulo Souza, Leonardo Borges, Piotr Luszczek, Stanimire Tomov, Jack Dongarra, Hartwig Anzt
prps@petrobras.com.br, leonardo.borges@intel.com, luszczek@eecs.utk.edu, tomov@cs.utk.edu, dongarra@cs.utk.edu, hanzt@icl.utk.edu
Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Ichitaro Yamazaki, Jesus Labarta
mgates3@utk.edu, haidar@eecs.utk.edu, yjia@utk.edu, kkabir@eecs.utk.edu, iyamazak@utk.edu, jesus.labarta@bsc.es

***Abstract*—This paper introduces a new heterogeneous streaming library called hetero Streams (hStreams). We show how a simple FIFO streaming model can be applied to heterogeneous systems that include manycore coprocessors and multicore CPUs. This model supports concurrency across nodes, among tasks within a node, and between data transfers and computation. We give examples for different approaches, show how the implementation can be layered, analyze overheads among layers, and apply those models to parallelize applications using simple, intuitive interfaces. We compare the features and versatility of hStreams, OpenMP, CUDA Streams[1] and OmpSs. We show how the use of hStreams makes it easier for scientists to identify tasks and easily expose concurrency among them, and how it enables tuning experts and runtime systems to tailor execution for different heterogeneous targets. Practical application examples are taken from the field of numerical linear algebra, commercial structural simulation software, and a seismic processing application.**

I. Programming in a Heterogeneous Environment

Effective concurrency among tasks is difficult to achieve, particularly on heterogeneous platforms. If this effort were more tractable, more people would tune their codes to achieve efficient performance. Our proposed hStreams framework makes it easier to port and tune task-parallel codes by offering the following features:

- **Separation of concerns**: The hStreams interface addresses real-world programmer productivity concerns, by allowing a separation of concerns between 1) the expression of functional semantics and exposure of task concurrency, and 2) the performance tuning and control over how tasks are mapped to a platform. Creators of scientific algorithms who want to harness computing resources are generally not computer scientists; they want something simple and intuitive. Code tuners and runtime developers may work long after the original scientific developers have moved on from their creations, and they tend to want the freedom to control *how* code executes without acquiring application domain expertise.
- **Sequential semantics**: Many users find a valid sequence of task invocations more natural to express than providing a dependence graph of concurrent tasks. Our hStreams library offers a sequential FIFO stream abstraction to make concurrency more tractable and easier to debug.
- **Task concurrency**: The concurrency that we focus on for this work is among tasks. It is orthogonal to issues like code scheduling, vectorization, and threading, all of which are important optimizations to apply within tasks.
- **Pipeline parallelism**: Platforms with distributed resources tend to have significant communication latency and constrained bandwidth, that needs to be overlapped; pipelining the transfer of one tile of data while computing on another tile is often critical to performance.
- **Unified interface to heterogeneous platforms**: When frameworks like OpenMP require users to handle task execution differently on local or remote resources, they increase the burden on programmers. hStreams, in contrast, offers a uniform task interface, to ease that burden.

The contributions of this paper are as follows: 1) the hStreams library framework, and a demonstration of its **applicability to heterogeneous platforms**; 2) a **comparison** with other programming models and language interfaces; 3) a description of our approach to layering hstreams above other plumbing layers and below other interfaces in commercial codes and academic frameworks, with **minimal overheads**; 4) an **evaluation** of hStreams on relevant kernels and applications, for different platforms, and with respect to NVIDIA CUDA Streams.

II. The hStreams Library

The hStreams[1] library provides a streaming, task queue abstraction for heterogeneous platforms, similar to CUDA Streams® [2] and OpenCL. The hStreams library focuses on heterogeneous portability. It has now been open sourced on github; see https://github.com/01org/hetero-streams/wiki. It was previously distributed with the Intel® Many-core Platform Software Stack, versions 3.4 to 3.6.

Features

The hStreams library manages task concurrency across one or more units of heterogeneous computing resources that we call *domains*. It uses queues called *streams* to localize the dependence scope and to offer a FIFO (first-in, first-out) semantic. Memory is managed across domains using an abstraction called *buffers*, which are used to help manage properties and track dependences. These three component building blocks offer abstractions that enhance programmer productivity, provide transparency and control, and enable a separation of concerns between the scientist programmer and the one tuning for a target architecture.

A **domain** is a set of computing and storage resources which share coherent memory and have some degree of locality. Examples of domains include a host CPU, a Knights family co-processor card, a node in a cluster reached across the fabric, a GPU, and a subset of cores that share a memory controller. Domains are discoverable and enumerable to users. Each domain has a set of properties that include the number, kind and speed of hardware threads, and the amount of each kind of memory, e.g. high-bandwidth memory.

[1]* Other brands and names are the property of their respective owners.

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.217

Streams are task queues across which task parallelism in achieved. Streams have a **source** endpoint, from which actions are issued, and a **sink** endpoint, at which actions occur. The sink is bound to computing resources identified by a domain and a CPU mask. The source and sink may be in the same domain (like OpenMP and TBB), or in different domains, such that work can be enqueued and scheduled on one set of resources and executed on a disjoint set of resources, if desired (unlike OpenMP). This provides a uniform interface for distributing work to computing on the local host or remote resources is more flexible and easier to use on hetero platforms than other interfaces. Three types of **actions** get enqueued into streams: *compute tasks*, *data transfers*, and *synchronizations*. They are free to execute and complete out of order, as long as the effect of such optimizations is not visible at the semantic level, *i.e.*, they do not violate the sequential *FIFO semantic* of the stream. Tasks naturally expand across a stream's threads when they use threading constructs like OpenMP or TBB.

Buffers encapsulate memory. They are used to **manage storage properties** (e.g. memory type and affinity) and **track dependences** among actions. All memory that can be referenced by user code is represented in a unified **source proxy address** space, which is partitioned into buffers. The virtual address of the base pointer of the buffer is stored for each domain in which the buffer is instantiated, so when an operand of an action associated with a stream falls within that buffer, its addresses are easily translated from the source proxy address to the virtual address needed for that stream's domain.

These three abstractions are useful for high-level programmers to specify semantics of actions and properties of storage. Those who perform tuning are left free to flexibly map these abstractions onto underlying physical realities in ways that optimize performance. The universality of allowing tasks to be executed on any subset of available domains makes hStreams fully retargetable; code can be added for new kinds of computing resources over time. The ability of tuners to define their own domains allows performance to be tuned for locality and enables portability. Users can expose concurrency across streams, but leave tuners with the responsibility for deciding which kinds of resources can best execute the actions of the stream (e.g. CPU or FPGA), and how many. The user's task naturally expands to use all of the resources given to a stream. For example, an `OpenMP for` in a task will use all threads assigned to that stream, and the code for the task need not change if the number of threads assigned to that stream changes, or the total number of threads available to divide among streams changes (e.g. between a 61-core Intel® Xeon Phi™ coprocessor (we henceforth call this **MIC**, for Many-Integrated Core) 7120P, a 57-core Xeon Phi 3110P, or a 12-core Xeon). Users may create a stream to operate on each tile, but the tuner can map multiple streams onto a common set of resources. Buffers give users a way to declare usage properties, such as whether it's read only and what its access patterns are, but give tuners control over the type of memory the data is bound to, how it should be affinitized, and how it should be managed in the memory hierarchy.

Compute, data transfer and synchronization actions can specify **memory operands**. These memory operands are the basis for data dependence analysis. In the current implementation, the user is responsible for explicitly moving data to domains in which it is needed. If a given sequence of actions within a stream has non-overlapping memory operands, then the runtime is free to execute and complete those actions out of order. For example, if compute task A is enqueued, followed by a transfer of data for independent task B, then B's data transfer may proceed out of order, concurrent with the execution of task A. Actual dependences among actions within a stream are implicitly specified by their FIFO order and their memory operands, and they are faithfully enforced. There are no dependences implied among actions in different streams, or between actions in streams and the source; those must be explicitly specified using synchronization actions.

To make best use of task parallelism, an application domain scientist would need to factor their code as a sequence of task invocations, where the heap structures that are used by the tasks are passed as parameters. A tuner can then follow up by wrapping those heap structures in buffers, and binding the tasks to streams. If the dependence predecessor of a given task is not in the same domain, a data transfer needs to be added. If the predecessor is in the same domain but a different stream, a synchronization action is needed. Otherwise, the FIFO semantic will manage the dependences within a stream implicitly, based on the buffer-wrapped operands. For examples see [1].

High-level hStreams APIs[1] allow the specified or visible (via automatic discovery) resources to be evenly divided up among a specified number of streams. Again this division and assignment can be under full user control with low-level APIs, or almost fully-automatic, with high-level APIs.

Usage examples

Several application and framework developers find that a **FIFO streaming abstraction** for tasks is useful, and they want to write to a **target-agnostic API**, and then map from that API down to several target-specific back ends. They have been able to do this for CUDA Streams on NVidia and OpenCL on AMD and perhaps they use OpenMP on a Xeon host. But without hStreams, they could not do this for a MIC coprocessor unless they changed their interfaces or wrote lots of code. The hStreams library makes this relatively easy to do. Several software vendors from the manufacturing space have ported from CUDA Streams (e.g. MSC[3]) and OpenCL (e.g. Simulia) to hStreams; early evaluations led them to productize with hStreams. Petrobras is an Oil and Gas customer with a similar API that is evaluating the hStreams library. OmpSs [4] is a data-flow programming model, based on OpenMP, that seeks to ease application porting to heterogeneous architectures. It exploits task level parallelism and supports asynchronicity, heterogeneity and data movement. The OmpSs team found several advantages to using hStreams over CUDA Streams, which are documented in the next sections.

III. Layering

Useful technologies often get sandwiched in the middle of several layers. The hStreams library is used in some frameworks as a plumbing layer, where completely encapsulating lower layers enables retargetability. Higher layers are simpler and more powerful, yet the abstraction of lower

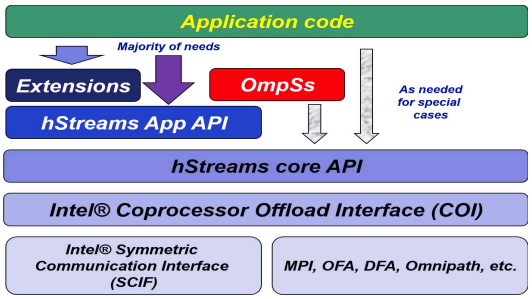

Fig. 1. hStreams APIs layered between fabric interfaces & higher abstractions.

layers can remain efficient, as we demonstrate below. One such lower layer used by Intel developer tools is the Intel® Co-processor Offload Infrastructure (COI), which optimizes platform-specific trade-offs. COI is in turn built on lower abstractions. In the PCIe context, it uses the Symmetric Communications Interface (SCIF) which abstracts low-level network hardware. COI supports offload over fabric, and could be built on top of MPI, TCP, Omni-path, PGAS, and proprietary interfaces. We exercised hStreams running on top of COI between Xeon nodes, but don't report results since this COI feature is still in development.

One of the frameworks that uses the hStreams library as a plumbing layer is **OmpSs**[4], which is described below. The hStreams library has also been used underneath proprietary, target-agnostic streaming APIs from several manufacturing and seismic application developers, including Simulia and Petrobras, that use CUDA Streams to target NVidia GPUs and OpenCL to target AMD GPUs. This is illustrated in Fig. 1.

In the evaluations throughout the paper, the machine configurations used for the Xeon (Haswell and Ivybridge) host and Knights Corner (KNC) co-processor are shown in Fig. 2.

hStreams' **performance overheads** are less than 5% for data transfers above 1MB. It has 20-30us of overhead for transfers under 128KB. For both hStreams and OmpSs, overheads for SCIF and hStreams on the host were negligible. The COI overheads are negligible when a pool of 2MB buffers were used. When they were not enabled, as in the OmpSs case, the COI allocation overheads were significant. The MIC-side overheads for hStreams and SCIF were all either just at initialization time or were negligible, as in the case of invocation overheads. OmpSs ends up inducing overheads on top of hStreams of 15-50% for matrices that are 4800-10000 elements on a side, as a cost of the conveniences it offers, as also shown in the Cholesky results of Fig. 7.

Fig. 2. Machine configuration.

Specification	Intel Xeon Processor E5-2697v2 (IVB) and E5-2697v3 (HSW)	Intel Xeon Phi Coprocessor C0-7120A (KNC)	NVidia K40x
Skt,Core/Skt,Thr/Core	2S,12C(v2),14C(v3),2T	1S, 61C, 4T	1S, 15C, 256T
SP, DP width, FMA	8,4,N (v2) 8,4,Y (v3)	16,8,Y	192, 64, Y
Clock (GHz)	2.7(v2) 2.6(v3)	1.33 (turbo)	0.875 (turbo)
RAM (GB)	64 DDR3-1.6GHz	16 GDDR5	12 GDDR5
L1 data, instr (KB)	32,32	32,32	64
L2 Cache (KB)	256	512	roughly 200
L3 Cache (KB)	32K(v2),35K(v3) (sh)	-	-
OS, Compiler	RHEL 6.4, Intel 16.0	Linux, Intel 16.0	-
Middleware	MPSS 3.6	MPSS 3.6	CUDA 7.5

IV. Comparison With Other Approaches

This section compares heterogeneous streaming approaches. It highlights the benefits of using a **library**, unlike Intel Compiler Offload Streams, and of providing a **uniform interface** to heterogeneous platforms, unlike OpenMP and CUDA Streams. hStreams makes it easy to achieve both **concurrency** among tasks across and within heterogeneous platforms, and thread parallelism within tasks. We begin with a comparison with OmpSs, which can be layered on top of hStreams or CUDA Streams. We highlight differences among these approaches in how they manage data, resources and execution flow.

OmpSs on top of hStreams

OmpSs [4] is a task-based programming model that enables sequential applications to run in parallel. Based on compiler directives to express tasks and data needed by those tasks, the framework is able to dynamically detect task data dependencies to exploit the inherent parallelism of applications. OmpSs supports combining several architectures in either single node or clusters [5], like CPUs, GPUs and MIC. In this work, the OmpSs runtime has been ported on top of hStreams to offload tasks to MIC cards and it provides a higher level abstraction layer for high programmer productivity, as described below.

Data management: OmpSs allocates data automatically on the device. Data movement between domains are implicitly added as needed for scheduled tasks. The runtime keeps track of data accesses by tasks to ensure program correctness.

Resource management: The runtime is in charge of transparently managing hStreams devices and performing all the needed actions, like creating and managing streams and events.

Execution flow management: OmpSs uses several streams and partitions to distribute work among system resources and maximize hardware utilization. Data is prefetched to the appropriate memory space as soon as possible and overlapped with other data transfers and computations. Internally, operations are always issued asynchronously with hStreams. This approach is also used for offloading tasks to GPUs and it has been shown to improve runtime performance [6]. Finally OmpSs is able to simultaneously offload tasks to CPUs, GPUs and MIC cards in heterogeneous platforms.

hStreams vs. Offload Streams

Offload Streams is a new feature of the Intel compiler[7] that builds upon Intel's past Language Extensions for Offload (LEO), independent of OpenMP directives on the host. The `stream` clause is added to the `offload` directive, and new API calls create, destroy and wait on a stream. There are differences among ways to enforce dependences between actions. While OpenMP uses the `depend` clause to call out which variables dependences are enforced on, Offload Streams uses `signal` and `wait` clauses. Dependent operands are explicitly listed as arguments to hStreams APIs for compute, transfer and synchronization actions. Both hStreams and the Intel Compiler implementations use COI.

Offload Streams supports streaming via offload to other devices only. It does not provide convenience functions that automatically create streams across available devices, or a mix of different device types. As a compiler feature, Offload Streams is able to support general parsing and type analysis of variables that are used as operands within offloaded code, and to identify the code to be offloaded and generate binary

code for the designated target. Their features are ubiquitously available to all those who pick up the latest compiler version. Clearly, for some users, keeping pace with the latest compiler is so time consuming that they prefer to change compiler versions infrequently, and to use library-based interfaces instead, even if that may affect invocation and code provisioning.

OpenMP

OpenMP added support for heterogeneous computing in its 4.0 spec[8]. Unlike other language extensions, OpenMP already boasts support for multiple heterogeneous architectures from a growing number of compiler vendors, making it a very attractive option for many programmers. Therefore we consider it important to have a comparison between OpenMP and hStreams. The comparison of OpenMP and hStreams here highlights their relative strengths, and reveals some opportunities where hStreams can fill some of the current gaps with respect to generality on heterogeneous platforms.

While OpenMP provides support for many different kinds of parallelism (e.g., loop parallelism and task parallelism) we restrict our comparison with hStreams to the heterogeneity support and affinity controls of both approaches. We compare with both the previous specification, OpenMP 4.0[8], and the OpenMP 4.5 spec[9], which addresses certain limitations of OpenMP 4.0 with respect to heterogeneous systems, most notably asynchronous data transfers. Unfortunately, a complete implementation of OpenMP 4.5 is *not yet available* for performance comparison, from either Intel or gcc/gomp.

The most obvious difference is that OpenMP is a **standard, compiler-supported language extension** while hStreams is a library-based API extension. The previous section highlights the key differences between a compiler and a library-based approach. We again focus our comparison on the same three key aspects of heterogeneity as above.

Data management: Both hStreams and OpenMP explicitly move data from one memory space to another. While hStreams buffers currently need to be allocated before the data can be transferred, an OpenMP allocation can happen either explicitly or implicitly. The hStreams allocation APIs support **allocation for different memory types**, e.g. for high-bandwidth or persistent memory, whereas OpenMP does not. OpenMP 4.5's support for asynchronous transfers closes a gap between hStreams and OpenMP 4.0.

Resource management: is a key source of differences between the two models. In hStreams, all resources are treated uniformly, whether they are on the host, a card, or a remote node. In OpenMP, there is a **clear separation in the constructs** used to create work in the **host and the devices**. The current hStreams implementation allows the creation of streams on devices residing in remote nodes (i.e., over fabric). hStreams provides the service of creating concurrent streams of arbitrary width, comprised of affinitized threads within a device. While an OpenMP implementation could theoretically sub-partition nodes into devices, we are not aware of an OpenMP implementation having productized this. OMP_PLACES, added in OpenMP 4.0, allows users to specify a list of thread numbers on which threads in a parallel region (such as the master thread of each stream) should be placed. Apparently only Intel's OpenMP implementation allows affinity to be manipulated within the execution of a program, and our users claim that using that mechanism is awkward compared to describing how many streams to create with hStreams' "app APIs", or to explicitly providing a mask for each stream with our "core APIs"[1].

Execution flow management: In both models, work is offloaded from a host thread to one of its resource abstractions (a logical device in OpenMP, a stream in hStreams). In hStreams, work offloaded to the same stream generates an implicit dependence in the execution order while in OpenMP work offloaded to the same device is independent by default. OpenMP 4.5 extends the device constructs to support the tasking synchronization primitives. Using this bulk synchronization (`taskwait` and `taskgroup`) and point-to-point synchronization, using the `depend` clause, allows synchronization of work and data transfers done in the host or in any of the attached devices, as long as they are at the same nesting level. But that **nesting constraint** poses an ease of use problem. Part of the value offered by OpenMP's dynamic task scheduler is that the scheduler does the binding to the target instead of the user. But if a task that does async offload is scheduled, and some other task depends on it, the only way to enforce that dependence is to make sure that the (parent) task that does the offloading completes. That synchronous enforcement of a transitive dependence (see [1]) can be costly. The alternative that avoids nesting is to have the user mark the task for offload with an `omp target` directive[1], but that defeats the value claimed above. In hStreams, additional APIs are provided for inter-stream synchronization and host to stream synchronization. In both cases, dependencies are based on a data-flow approach where actual dependencies between work units are derived from the declared input and output operands of the task.

In summary, the major differences between OpenMP and hStreams is OpenMP's lack of ability to **subdivide a device** to be able to have multiple offload regions running concurrently onto **disjoint sets of heterogeneous resources**, and to **span multiple devices in a uniform way**. The performance data that we will gather upon OpenMP 4.5 completion will support the need for OpenMP to look in this direction.

CUDA Streams

Streaming abstractions may differ in their usability and capabilities. Several differences between hStreams and CUDA Streams[2] are highlighted below. These differences became clear from our comparative implementation of both hStreams and CUDA Streams within the OmpSs framework.

Data management: Especially when dealing with multiple devices, maintaining all of the device-side addresses in CUDA can be complicated, as every device may have its own address space. Then, multiple variables are needed to keep the addresses for each memory space. With hStreams, only the host proxy address is used in the application's source code, so the code looks **cleaner and is less error prone**. hStreams allows a single host proxy address to represent (e.g., shared) instances in many domains; CUDA does not.

Resource management: Unlike CUDA Streams, hStreams allows the possibility of dividing the computing resources into smaller groups. This allows the programmer to adapt this division depending on application's concurrency to improve hardware utilization and load balancing. The use of a hardware scheduler in CUDA Streams may enable adapting to less concurrency than a configured number of streams.

Execution flow management: In CUDA, **explicit event and stream creation and destruction** is required. Streams in hStreams are represented by an integer in contrast to the CUDA opaque pointers that are returned at stream creation and need to be used to send operations to such streams. hStreams adds the possibility of **waiting on a set of events** and being signaled when one or all the events are finished. This can save CPU spinning time. Streams and partitions can be highly configured by the programmer using low-level APIs, but they can also be set easily with high-level APIs. CUDA Streams follow a strict FIFO order of operations, and are not pipelined, while hStreams actions in the same stream do not need to wait for previous operations to complete as long as they are data independent. This behavior can be implemented in CUDA Streams at the cost of using several streams and introducing synchronization points between them, which increases the complexity and programming effort. hStreams' flexibility can potentially increase an application's concurrency with no additional effort for the programmer. For a 4Kx4K matrix multiply in OmpSs, the hStreams-based implementation was **1.45x faster than CUDA Streams**. The primary contributors to the performance difference are that for CUDA Streams, OmpSs needs to explicitly compute and enforce dependences, whereas this is not necessary within hStreams.

Source code: hStreams does not require writing **device-specific code** when targeting MIC. Since the same code can also be run on the host, it is more portable and programmable. CUDA's special syntax requires the use of the nvcc compiler. hStreams applications can be compiled with any compiler.

Sample API Comparison: Fig. 3 presents a summarized comparison of hStreams and the other approaches evaluated in this section for matrix multiply. In the top part, the number of *additional* source code lines to perform offload is shown for each of the different application phases, for several language interfaces. The central part of the figure shows the number and size of additional support variables needed for both hStreams and CUDA versions, when explicit synchronization is used. The bottom part counts the number of API calls needed for each approach and the achieved performance. A table showing the actual code comparison is available at Ref. [1, 10]. Performance results are shown at the bottom of the table. A fully-functional OpenMP 4.5 compiler is not yet available, and we did not have a CUDA formulation of this code. hStreams requires far fewer extra lines of code, API calls and unique APIs than CUDA (1.94x-2.25x) and OpenCL (1.65x-2.00x). OpenCL performance is poor because cuBLAS is not well tuned for MIC. The OpenMP 4.0 has one extra API that does the allocation, transfer, invocation and deallocation, but OpenMP does not use concurrency within the device and does not support an asynchronous transfer. So a tiled implementation has less than half of the performance: 180 vs. 460 GFl/s. hStreams provides the highest performance of the alternatives shown. It's not required that hStreams or CUDA Streams use events for this particular example, but they are illustrated there.

Fig. 3. Coding Comparison - see [1, 10] for actual code examples

# additional source code lines vs. basic tiled version						
Descr.	hStreams	CUDA	OMP 4.0	OMP 4.5	OmpSs	OpenCL
Initialization	2	9	0	0	0	8
Data alloc	3	6	0	3	0	6
Data transfers	7	7	0	7	0	7
Computation	0	2	1	1	3	0
Synchronization	1	1	0	1	1	1
Data transfers	2	2	0	2	0	2
Data dealloc	3	6	0	3	0	6
Finalization	2	7	0	0	0	3
Total	20	40	1	17	4	33

# support variables	
hStreams	CUDA
1 matrix[M][N][L] (events)	1 matrix [M][N] (streams) 1 matrix [M][N][L] (events) 1 opaque pointer (CUBLAS) 1 matrix [M][L] (dev. addr.), 1 matrix [L][N] (dev. addr.), 1 matrix [M][N] (dev. addr.)

Metric	hStreams	CUDA	OMP 4.0	OMP 4.5	OmpSs	OpenCL
Unique APIs	8	18	1	5	5	16
Total APIs used	16	31	1	14	9	28
GFl/s,$(10K)^2$	916	N/A	460, 180	N/A	762	35

Other Related Work

The hStreams interface does not require OpenCL's boilerplate code. It does not rely on target-specific tuning for clBLAS [11], which is significantly under-optimized for the MIC. Unlike tasks in SyCL [12], OpenMP, MPI, OmpSs [4], StarPU[13], Qualcomm MARE[14], CnC[15], OCR[16], TBB Flow Graph[17] and others, task dependences need not be explicitly specified. Unlike SyCL[12], Phalanx[18], CnC[15], UPC++[19], CHARM++[20], TBB Flow Graph[17, 21], Kokkos[22], Legion[23] and Chapel[24], there is no current dependence on C++ or other language mechanisms. As a library, hStreams doesn't depend on users to stay current with the latest versions of compilers, as is the case for the Intel compiler [25]'s Offload Streams for and OpenStream [26]. Furthermore, it doesn't need to "own main", as OCR[16], CHARM++[20] and others do, for the sake of initialization, e.g. to set up communication. The current hStreams interfaces are at a lower level than related systems such as Trilinos[27] and Legion[23]. Other frameworks that deal with distributed environments, like Global Arrays[28], HPX[29] and TIDA[30] don't offer the FIFO stream abstraction of hStreams. hStreams offers a distinction between logical and physical abstractions not available in LIBXSTREAM[31] or icc[25]. hStreams does not yet automate dynamic scheduling, as TBB Flow Graph, Legion, CnC, HPX and others do.

V. Applications

In this section, we discuss some applications to which hStreams has been applied and the associated algorithms. We look at how algorithms are modified to support tiling, and how work can be distributed to multiple domains to optimize for concurrency. In the following we will first describe in detail two fundamental reference applications, matrix multiplication and Cholesky factorization, followed by two commercial applications, Simulia Abaqus/Standard and Petrobras RTM.

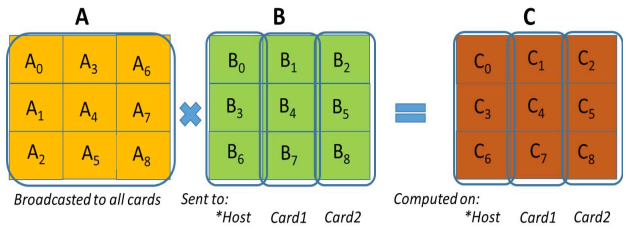

Fig. 4. Decomposition of matrices into tiles and distribution of tasks onto the host and multiple cards for the hetero hStreams matmult program. **Host* refers to host-as-target streams.

Matrix Multiplication

Matrix multiplication is an expensive mathematical operation used in many software applications. Here we describe a manual mapping onto to host and multiple MICs using hStreams. This builds upon the single card offload program given in Ref. [32]. For this algorithm, the matrices **A**, **B**, and **C** are divided into square tiles as shown in Fig. 4. Matrix **A** is broadcast, one tile at a time, to both the host (using host-as-target streams) and all MICs. Matrix **B** is partitioned into column panels containing one or more tiles; these tiles are sent one tile at a time either to the host or to a MIC. The number of panels is chosen as an integer multiple of the number of MICs plus one (host). Transfers to the host in host-as-target streams are optimized away. Matrix **C** is also partitioned into panels; each panel is assigned to a unique computational domain (either host or card(s)) responsible for its update. Panel updates are independent and do not require communication between MICs. The combination of tiling and multiple streams allows the design of effective strategies to hide the latency associated with data transfers; C-panel computations may start as soon as a few tiles arrive at the MICs as opposed to the traditional offload approach where the computation cannot start until the entire matrices reside on the MICs.

Cholesky Factorization

Cholesky factorization is a decomposition of Hermitian, postive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, useful for efficient numerical solutions of linear systems arising in a large number of scientific fields. To exploit the functionality of the hStreams library in hiding data transfer latencies and improving concurrency, the input matrix is divided into tiles as shown in Fig. 5. The native, host version, of the tiled-Cholesky factorization algorithm employed in this paper is given in Ref. [33]. Fig. 5 shows different compute operations (LAPACK and BLAS functions) which are applied on different tiles. Intel® MKL is used for these function calls. The colors show computes in the first pass of the algorithm. In the algorithm, one moves from column to column starting from the leftmost column and the colors (and operations) are shifted right each time. The tiles on which DPOTRF and DTRSM have been applied contain the final factored data for the column.

The manual hetero algorithm using hStreams for tiled-Cholesky builds upon the single MIC card offload program given in Ref. [32]. For this, DPOTRFs, DTRSMs and some of the DSYRKs and DGEMMs execute on the host (in host-as-target streams). For DPOTRF, we use a machine-wide stream

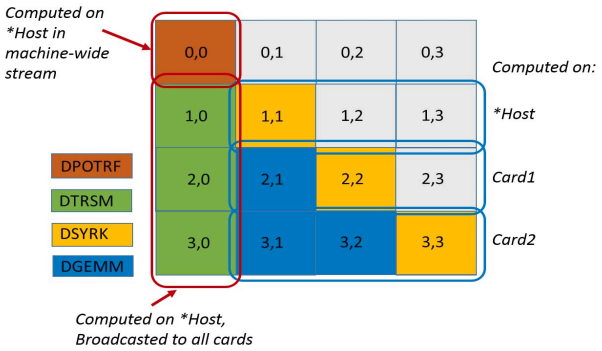

Fig. 5. Decomposition of the matrix into tiles and distribution of tasks onto the host and multiple cards for the hetero hStreams Cholesky program. **Host* refers to host-as-target streams.

on the host. The results of DTRSMs are broadcast to all cards. Each tile-row is assigned to either the host or one of the cards in a round-robin fashion. Each subsequent compute on the host or a given card is round-robin'd across the available streams on that computing domain. The results of DSYRK and DGEMMs in the column adjacent to the DTRSM column are sent from different cards to the host. This process repeats in the next pass. Note that since there are no dependencies between DSYRK and DGEMMs and since each card computes on a fixed row, no data card-card transfers are needed. Each card only interacts with the host. Once again, any transfers en-queued in host streams are aliased and optimized away.

MAGMA

The MAGMA (Matrix Algebra on GPU and Multicore Architectures) [34] provides a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures. MAGMA offers a version that targets MIC. The lower Cholesky MAGMA function uses the host for the DPOTRF panel and does the rest of the work on the MIC card. We will compare our hStreams Cholesky performance against MAGMA Cholesky performance in the next section.

Simulia Abaqus/Standard

Simulia's Abaqus/Standard software product[35] simulates static and low-speed dynamic events where highly accurate mechanical stress solutions are critically important. Examples include sealing pressure in a gasket joint, steady-state rolling of a tire, or crack propagation in a composite airplane fuselage. They accelerate highly-parallel computations on AMD GPUs with OpenCL, on NVidia GPUs with CUDA Streams, and now on MIC coprocessors with hStreams, all as back ends for a target-agnostic API that their developers code to. The initial release of **Abaqus 2016 uses hStreams** in the symmetric solver; coverage for unsymmetric solver will follow in the first maintenance release. The symmetric solver that is evaluated in this paper is related to the hStreams Cholesky reference code in the following way. It uses similar factorization: $\mathbf{LDL}^T$ instead of $\mathbf{LL}^T$. As a research topic, Simulia is investigating the use of the host for streaming computations using hStreams, in addition to offloading to multiple cards. For this purpose a standalone test program is developed that factorizes a single

dense supernode. The full production solver processes all of the supernodes in a given system of equations in an optimized order. We will show the performance results for the full solver as well as for the standalone test program in the next section.

Petrobras RTM

Reverse Time Migration [36] is heavily used by Petrobras in seismic processing, on HPC clusters with thousands of nodes. A single job can run from weeks to months. The core of RTM is usually a time domain finite difference wave field propagator [37], a stencil-based computation [38], where each point reads from its neighbors in a 3D regular grid.

Petrobras has a high-level Fortran90 library called **HLIB** [39] that abstracts three back ends: CUDA, OpenCL and the CPU. Although there's one architecture with optimized implementations for each back end, all the device management needed is done with a high-level target-agnostic Fortran API.

A production-size grid won't fit in one co-processor memory, so a domain decomposition into subdomains is needed. At every wave propagation step each accelerator needs to exchange information with its neighbors, since every point in the grid needs information on its neighborhood. There's a distinction between two kinds of grid points within each subdomain: halo points, that need to produce data for neighboring subdomains, and internal points, that do not need to exchange data that way in this time step. Processing of data for halo grid points should be given priority, so that the internal grid points can be processed while the data exchange is occurring.

A streaming abstraction fits well here. In the simplest form, halo grid points are processed in one (set of) streams, and interior (bulk) grid points are processed in another (set of) streams. Data exchange with neighbors can be initiated once all of the processing in the first set of streams is done. In such a scheme, there's an implicit barrier, waiting for all actions to complete in the first set of streams before any data exchanges start. That scheme can work fine, if the work on the bulk points takes longer than both the work on the halo points and the cost of the data exchange. When that condition holds, having data transfers complete in FIFO order, and not having any special data dependence management is fine. CUDA Streams provides that, and this is what the existing Petrobras implementation does. Being able to have hStreams plug into Petrobras' target-agnostic streaming API enables it to be ported quickly to heterogeneous clusters.

But there's an alternate scheme, that becomes increasingly relevant given the trend towards increasing communication costs on large grids, that may eventually exceed the cost of processing the interior grid points. That can happen when the ratio of halo points to interior points increases, because the subdomain is small, e.g. to fit in constrained memory, or when dealing with very high-order stencils. In that case, we don't want to wait until all halo grid points are processed to start exchanging data. Instead, we want to queue up transfer tasks that depend on the completion of processing of each halo grid point. That involves making those data transfer tasks dependent on the compute tasks. A FIFO stream semantic can save the user from having to make those dependences explicit. But if the transfers are to be concurrent with the computes, they'd need to be in different CUDA Streams, and explicit synchronization would need to be added, since actions are in different streams. This is unnecessary with hStreams. And what if there's load imbalance across the tasks, and some finish earlier than others? In that case, the FIFO *semantic* promotes ease of use, but a FIFO *implementation* would inhibit performance. Unlike CUDA Streams, hStreams enables a FIFO semantic and an **out of order execution**. Thus the data exchange tasks (if MPI is used explicitly as is done at Petrobras today) or even direct, enqueued data transfer actions (if hStreams is used for communication plumbing), can be queued into the same stream as their related computation, and they can get dynamically scheduled, as early as the data is ready. This accommodates 1) slow communication, 2) more-dominant communication costs when the ratio of halo points to interior points is high, due to small sub-domains or high-order stencils, and 3) dynamic load imbalance. These two schemes are evaluated and compared below.

VI. Performance Results

Good performance on heterogeneous platforms is a result of a careful choices of where tasks run most efficiently, of how to maximize concurrency across and within nodes in the platform, and of keeping overheads low. We start with concurrency considerations within a node, then look at distributing tasks across a heterogeneous platform.

Within a Node: Tiling, Concurrency, Balancing

Tiling is the process of decomposing large matrices down into smaller ones, and modifying an algorithm to work on the tiles instead of the monolithic matrices. Decomposition increases concurrency. Tiling can help in that a smaller amount of data needs to get transferred to a computing resource before the work can begin on it. Some or all of the communication latency can get covered by computation, through pipelining. Decomposition creates a larger number of tasks, which may be more evenly divisible by the number of computing resources, leading to less load imbalance.

The best degree of tiling and number of streams depends on the matrix size and algorithm. Users want to be able to tune these easily, by changing just a few parameters. We provide a detailed analysis of that technique in [32] for matrix multiply, Cholesky, and LU, but only for offloading to a single KNC card rather than multiple MICs or MICs plus a host.

Distributing Tasks on a Heterogeneous Platform

Compute efficiency is straightforward. When a given task is run on different computing elements, where does it run best? At present, DGETRF runs better on the host than the coprocessor, and an untiled scheme works best for sizes smaller than 4K [32]. An untiled Cholesky (DPOTRF) runs better natively on a Haswell (HSW) Xeon than on coprocessor for matrix sizes up to 20,000 (see Fig. 7). The trade-off is more complicated: 1) should it be decomposed into tiles and 2) should DPOTRF run on the host or coprocessor, as it also affects the number of data transfers?

Fig. 6 shows matrix multiplication performance on a hetero platform. Up to 2599 Gflops/s is achieved for a HSW node

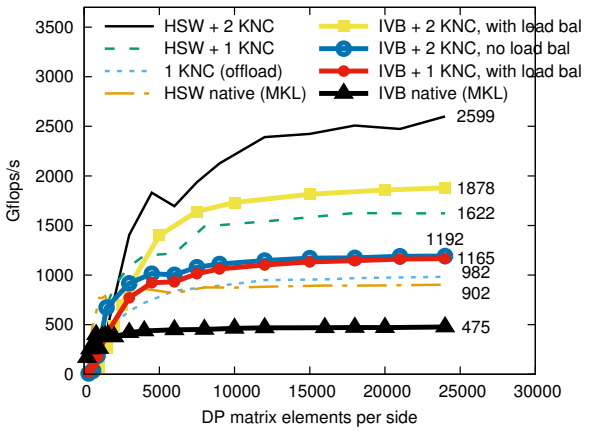

Fig. 6. Performance of hetero hStreams Matrix-Multiply for different platforms and configurations.

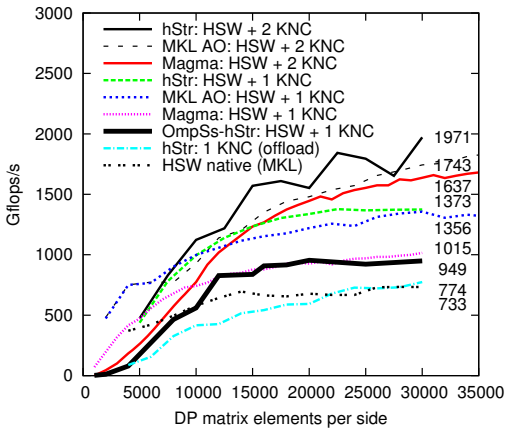

Fig. 7. Performance of Cholesky for different platforms and implementations: hStreams code (hStr), MKL Automatic Offload (AO), MAGMA, OmpSs.

with 2 KNC cards. The relative capabilities of the host and card affect the importance of load balancing. In a naive distribution of work across computing resources, the same number of tasks are given to each of the host and card. This is fine for Haswell, which has similar performance on DGEMM (902 GFl/s) to a KNC (982 GFl/s). But in the Ivybridge (IVB) case, where host performance drops to 475 GFl/s, performance with load balancing on 2 MICs (1878 GFl/s) is 1.58x higher than without (manual) load balancing (1192 GFl/s), see Fig. 6.

Another set of trade-offs pertains to how concurrency is exploited across and within nodes. Consider the distribution of tasks for a tiled Cholesky across multiple MICs, shown in Fig. 5. One possible distribution executes the DPOTRF and all three DTRSMs on the Xeon host, with results broadcast to all MICs, and the remainder of the lower three rows of DSYRK and DGEMM tasks to host and 2 MICs. This particular distribution maximizes concurrency across MICs and minimizes communication. As seen in the hStreams curve in Fig 7, the scaling efficiency degrades as the number of MICs is increased, because the tiles shown in grey above the diagonal in Fig. 5 don't need to be computed. For the tiled-matrix multiply, on the other hand, we see excellent scaling for 2 MICs (efficiency of $>$ 85% for matrix sizes $>$ 12000, for HSW host) because as shown in Fig. 4 there is perfect load balancing among MICs and a simple communication pattern.

The difficult trade-offs in decomposing and distributing tasks are shown to matter, so having a very flexible, intuitive and efficient framework in which to map concurrent tasks onto heterogeneous platforms, as we've found hStreams to be, can have a significant impact on productivity. Improving that productivity is key to achieving good performance.

We now examine the relative performance of different implementations for Cholesky factorization that use both the host and MIC cards: hetero hStreams code, MKL AO (automatic offload), MAGMA and OmpSs. The results are shown in Fig. 7, which compare results for the host only, with an MKL native DPOTRF call, and with the host plus 1 or 2 MIC cards. The fact that 10% greater performance was achieved with hStreams with four days of tuning to use the host rather than just purely offloading to multiple MICs, vs. months of development by the MKL team to develop and tune their AO infrastructure is an indication of both the power of hStreams' infrastructure and its ease of use. MAGMA performance when using a single KNC card exceeds that of hStreams' with KNC-only or HSW-only configuration because MAGMA code ships the DPOTF2 panel factorization back to the CPU and thus the MIC spends most of the execution time in much more efficient DTRSM, DSYRK, and DGEMM routines. However, hStreams outperforms MAGMA by 10% when the host and MIC are used because spare host resources are used for more efficient routines in addition to the latency-bound DPOTF2. Also compare the MAGMA's rather smooth performance curve with hStreams' noticeably-jagged performance, especially on two MIC cards. This is a result of the MAGMA's team effort in performance engineering around sporadic inefficiencies that may be present in the software stack (BLAS implementation from MKL, communication library, etc.) and its tailoring to the hardware configuration (cache line size, core count, etc.). hStreams itself is not focused only on linear algebra software; it does not apply domain-specific knowledge to dynamically adjust the parametric setting of the algorithm or switch the underlying numerical method for greater efficiency without loss of accuracy as MAGMA does. OmpSs has only been tested in offload mode and for only one MIC. The performance for sufficiently-large problem sizes ($>$12K) is the same as that of MAGMA even if not using the host for computational tasks. For small problem sizes, granularity issues and the overhead of OmpSs fully dynamic task instantiation and scheduling result in lower performance.

Simulia Abaqus/Standard

Fig. 8 shows the performance of several customer-representative workloads evaluated in the Simulia full application, which uses hStreams in its released production code. The 8 workloads are identified by name, with proprietary customer workloads assigned a letter: A, B or C. The test cases cover not only the common symmetric cases, but also unsymmetric cases. The speedups from adding the use of MIC cores via hStreams to the standard use of Xeon cores look promising. Speedups are shown relative to both the weaker IVB baseline (up to 2.61x for the solver kernel and 1.99x for the entire application) and the more-capable Haswell (HSW) (up to 1.45x and 1.22x, respectively). The latter obviously has

lower speedups, since the HSW peak compute performance is approximately twice the Ivy Bridge (IVB) performance, as can also be seen in Fig. 6. Only the solver is offloaded to the MIC cards. The difference in speedups obtained for the solver and the full application is dependent on how "solver-dominant" the workload is, as well as other initialization costs.

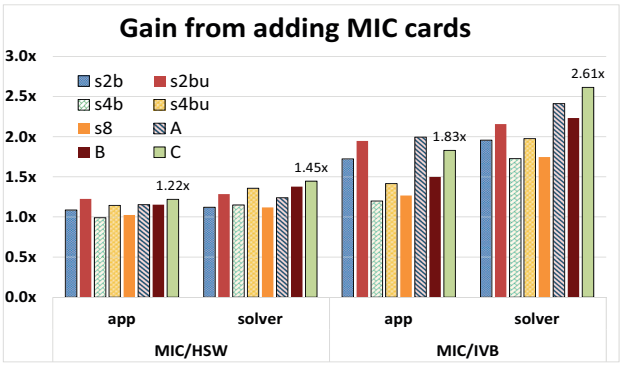

Fig. 8. Speedups for Abaqus/Standard when adding 2 MIC cards to Xeon cores. Data for 8 workloads for IVB and HSW host CPUs is shown.

To investigate the potential benefit of using host-as-target streams, we use a standalone test program which factorizes a single representative supernode, and show the run times in seconds for factorizing the supernode on KNC card, Haswell host CPU (HSW, 2 years newer than KNC), and Ivy Bridge host CPU (IVB, includes transfer time) in Figure 9. The relative run times correlate pretty well with the relative peak performance of these platforms. There is a plan to modify the standalone test program to use hStreams to make concurrent use of host and MIC streams in a unified programming framework in the future.

KNC offload	HSW host-as-target	IVB hvost-as-target
2.35	2.24	4.27

Fig. 9. Runtimes (s) for Abaqus standalone hStreams test program. 4 streams for KNC (60 threads each), 3 streams for HSW (9 threads each), and 3 streams for IVB (7 threads each) are used. The median of 5 runs is reported.

We compare the net effectiveness of parallelizing for a hetero platform, including host-target concurrency, streaming overheads and hiding communication costs, we **compare Simulia Abaqus performance for hStreams and CUDA Streams** implementations. To make this a fair comparison, the card-side work should be the same. This was achieved by using a single card on three representative standard workloads and the card-side FLOPS were confirmed to be within 1% for the two cases. This comparison is clouded by two issues: 1) the K40x is faster than the KNC, yet 2) card performance is not always on the critical path. When card performance is on the critical path, we can normalize with respect to card-side performance. This is assessed for KNC using VTune (sum of work and OpenMP times on all threads/240 threads), and for K40x using the sum of kernel times, as reported by nvprof. Since kernels may overlap on different SMXes, this is an upper bound, which *could lead to under-normalization in K40x's favor*. The results for [s4b, s8, "A"] show a [1.12x, 1.17x, 1.27x] solver advantage for K40x, but a [1.28x, 1.24x, 1.03x] advantage for KNC, respectively, when normalized to card-side performance. The middle of these ranges is within a couple percent of parity, suggesting that the two streaming approaches have comparable performance for radically-different targets, and hStreams holds promise for bringing out forthcoming performance improvements with Knights Landing.

Petrobras RTM

An initial implementation of Petrobras' MPI-based reverse time migration in hStreams shows promising results . We compare, for one to four ranks, 1) a baseline case which executes one rank on a HSW with no offload, 2) fully-synchronous offload of the highly-parallel code to a KNC card with no overlap of data and compute, and 3) asynchronous, pipelined-overlap offload to a KNC card. In the asynchronous, pipelined case (3), the MPI send and receive is executed on the host, the data movement for the upper and lower halo is pipelined with the upper and lower halo and bulk (non-halo) computation. We evaluated whether the halo tasks were small enough to benefit from task concurrency in multiple streams. Their workload of $1K \times 1K \times 8 * 80$ Flops is large enough to dwindle the OpenMP fork/join overheads and obviate the need for such concurrency. The benefit of asynchronous pipelining ranges from 3 to 10%. Performance tuning benefits KNC significantly: the speedup from using a KNC over just a Haswell host is 1.52x for 1 card and 6.02x for 4 ranks on 4 MICs for optimized code. For unoptimized code, the speedup, 1.13x-4.53x, is lower since communication is a smaller fraction and hiding it is less beneficial.

VII. Conclusions & Future Work

A streaming abstraction is one of several compelling choices for mapping concurrent tasks to heterogeneous platforms. We now summarize how the hStreams framework meets the goals for supporting heterogeneity with streaming.

- **Offload** to captive cards over PCIe: we show compelling performance for how hStreams is able to offload work to one or more MIC co-processors.
- **Portability** between pure offload to one or more cards and also streaming within the (host) node.
- **Effective layering** under Simulia Abaqus/Standard, OmpSs, and Petrobras in production environments, and over COI (co-processor offload interface)

Ease of use is a primary goal for optimizing large code projects deployed on heterogeneous clusters. Here's an assessment of how some goals for ease of use were met:

- Fewer lines of code ($[20, 40, 33]$ for [hStreams, CUDA Streams, OpenCL] in one case), fewer API calls ($[8, 18, 16]$ unique APIs, $[16, 31, 28]$ API instances), less variable allocation: the competitiveness of hStreams was illustrated in a complete example for matrix multiply.
- Lower overheads for cross-stream coordination, as illustrated by a 1.45x advantage for hStreams vs. CUDA Streams in OmpSs on a 4Kx4K matrix.
- Ease of design exploration, with respect to target affinity, changing degree of tiling and number of streams, and remapping parallelism across different nodes in a heterogeneous cluster was illustrated with examples like multi-card Cholesky, where hStreams outperformed MKL Automatic Offload and MAGMA for MIC.

- Ease of porting and future proofing through separation of concerns: Easy code porting across alternative heterogeneous configurations was illustrated on a simplified representation of Petrobras's RTM code.

Finally, a heterogeneous streams framework needs to achieve good performance.

- Enabling concurrency between nodes, within a device, between computation and communication: each of these were illustrated with matrix multiplication. Outperformed MAGMA and MKL AO by 10%, boosted Petrobras offload by 10%, yielded 2x gains over just a host.
- Less dependence analysis and enforcement work, leading to a 1.4x gain for OmpSs relative to CUDA Streams on a 6K x 6K matrix 2x2-tiled multiply
- Ease of performance tuning: the relevance and ease of varying the degree of tiling, number of streams and target affinitization was established for matrix multiply and Cholesky for hStreams, was explained.

In conclusion, hStreams meets all of these design criteria, and we've demonstrated these points with supporting data.

Petrobras is evaluating hStreams for future production deployments with Knights Landing. Simulia is considering applying hStreams to their Eigenvalue solver, and also their AMS solver, which currently depends on MAGMA for MIC. MAGMA is looking to rework some of its infrastructure to use hStreams. The overhead analysis performed as part of this work revealed that making MIC-side memory allocation asynchronous is a bottleneck; this feature is now forthcoming.

ACKNOWLEDGMENTS

We would like to acknowledge Paul Besl for representing customer interests for hStreams, the hStreams implementation team, Efe Guney for his analysis of MKL performance, Sumedh Naik for OpenCL help, and our reviewers for their constructive feedback. We thank Mallick Arigapudi, Michael Hebenstreit, and Danny Pacheco for assistance in running jobs on Intel's Endeavor cluster.[2]

REFERENCES

[1] See: https://01.org/hetero-streams
[2] See: docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
[3] "Multidiscliplinary structural analysis." See: www.mscsoftware.com/product/msc-nastran
[4] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a Proposal for Programming Heterogeneous Multi-core Architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.
[5] J. Bueno-Hedo, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive Programming of GPU Clusters with OmpSs," in *Proceedings of the 26th IEEE Int. Parallel and Distributed Processing Symposium*, ser. IPDPS 2012, 2012.
[6] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "AMA: Asynchronous Management of Accelerators for Task-based Programming Models," *International Conference on Computational Science, ICCS 2015*, vol. 51, no. 0, pp. 130 – 139, 2015.
[7] "Offload streams." See: software.intel.com/en-us/articles/intel-c-compiler-160-for-windows-release-notes-for-intel-parallel-studio-xe-2016
[8] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," July 2013. See: www.openmp.org/mp-documents/OpenMP4.0.0.pdf
[9] ——, "OpenMP application program interface version 4.5," Nov 2015. See: www.openmp.org/mp-documents/openmp-4.5.pdf
[10] See: pm.bsc.es/content/src-comp-hstr-others
[11] See: github.com/clMathLibraries/clBLAS
[12] See: www.khronos.org/opencl/sycl
[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.
[14] "Parallel computing (mare)." See: developer.qualcomm.com/mobile-development/maximize-hardware/parallel-computing-mare
[15] K. Knobe, "Ease of use with concurrent collections (cnc)," in *Proceedings of the First USENIX conference on Hot topics in parallelism*. USENIX Association, 2009, pp. 17–17.
[16] "Open Community Runtime Wiki." See: xstackwiki.modelado.org/Open_Community_Runtime
[17] "TBB Flow Graph Reference." See: www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm
[18] M. K. M. Garland and Y. Zheng, "Designing a unified programming model for heterogeneous machines," 2012.
[19] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick, "Upc++: A pgas extension for c++," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 1105–1114.
[20] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. ACM, 1993, pp. 91–108.
[21] "The Intel Threading Building Blocks Flow Graph," 2015. See: www.drdobbs.com/tools/the-intel-threading-building-blocks-flow/231900177
[22] H. C. Edwards and D. Sunderland, "Kokkos array performance-portable manycore programming model," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, 2012, pp. 1–10.
[23] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11.
[24] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.
[25] See: software.intel.com/en-us/intel-compilers
[26] A. C. Antoniu Pop, "Openstream: Expressiveness and data-flow compilation of openmp streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, 2013.
[27] M. A. e. a. Heroux, "An overview of the trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, Sep. 2005.
[28] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apr, "Advances, applications and performance of the global arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.
[29] "High Performance ParalleX." See: stellar-group.github.io/hpx/docs/html/
[30] "Tiling as a durable abstraction for parallelism and data locality," in *Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, 2013.
[31] See: github.com/hfp/libxstream
[32] *High Performance Parallelism Pearls*. Morgan Kaufman, July 2015, ch. Fast matrix computations on asynchronous streams.
[33] E. Jeannot, "Performance analysis and optimization of the tiled cholesky factorization on numa machines," *PAAP 2012-IEEE International Symposium on Parallel Architectures, Algorithms and Programming*, 2012.
[34] Innovative Computing Laboratory, University of Tennessee, "MAGMA." See: icl.cs.utk.edu/magma/
[35] "Abaqus standard." See: www.3ds.com/support/certified-hardware/simulia-system-information/abaqus-614/performance-data/
[36] P. Souza, T. Teixeira, L. Borges, A. Neto, C. Andreolli, and P. Thierry, "Reverse time migration with heterogeneous multicore and manycore clusters," in *EAGE Workshop on High Performance Computing for Upstream*, 2014.
[37] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood, "Reverse time migration," *GEOPHYSICS*, vol. 48, no. 11, 1983.
[38] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.
[39] P. Souza, C. J. Newburn, and L. Borges, "Heterogeneous architecture library," in *Second EAGE Workshop on High Performance Computing for Upstream*, 2015.

[2]Performance may depend on system configuration and usage. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on systems not manufactured by Intel. See http://www.intel.com/performance.