Mapping a Data-Flow Programming Model onto Heterogeneous Platforms

Alina Sbîrlea[†] Yi Zou[‡] Zoran Budimlić[†] Jason Cong[‡] Vivek Sarkar[†]

[†]Rice University
{alina,zoran,vsarkar}@rice.edu

[‡]University of California, Los Angeles
{zouyi,cong}@cs.ucla.edu

Abstract

In this paper we explore mapping of a high-level macro data-flow programming model called Concurrent Collections (CnC) onto heterogeneous platforms in order to achieve high performance and low energy consumption while preserving the ease of use of data-flow programming. Modern computing platforms are becoming increasingly heterogeneous in order to improve energy efficiency. This trend is clearly seen across a diverse spectrum of platforms, from small-scale embedded SOCs to large-scale super-computers. However, programming these heterogeneous platforms poses a serious challenge for application developers. We have designed a software flow for converting high-level CnC programs to the Habanero-C language. CnC programs have a clear separation between the application description, the implementation of each of the application components and the abstraction of hardware platform, making it an excellent programming model for domain experts. Domain experts can later employ the help of a tuning expert (either a compiler or a person) to tune their applications with minimal effort. We also extend the Habanero-C runtime system to support work-stealing across heterogeneous computing devices and introduce task affinity for these heterogeneous components to allow users to fine tune the runtime scheduling decisions. We demonstrate a working example that maps a pipeline of medical image-processing algorithms onto a prototype heterogeneous platform that includes CPUs, GPUs and FPGAs. For the medical imaging domain, where obtaining fast and accurate results is a critical step in diagnosis and treatment of patients, we show that our model offers up to 17.72× speedup and an estimated usage of 0.52× of the power used by CPUs alone, when using accelerators (GPUs and FPGAs) and CPUs.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Data flow model, Heterogeneous architectures, Domain-specific language, Tuning annotations.

1. Introduction

Energy efficiency is becoming a critical criteria for designing both embedded and large-scale computing systems [23]. The current trend in hardware design is to develop heterogeneous, massively parallel multicore architectures that can be specialized to satisfy the energy constraint. In embedded computing, modern SOCs typically integrate DSP processors and several domain-specific (such as video and audio) accelerators. In supercomputing, GPUs and massively parallel many-core accelerators are present in many of the top HPC systems[1]. GPUs can boost a very high computing capability with a smaller (per FLOP) energy demand, at a price of very low programmability.

These radical changes in hardware architectures are already impacting software, as the burden shifts to the programmer to take advantage of the available parallelism. The additional software complexity forces the rethinking of the design of computer systems. Old views have already begun to change, moving from mapping software on complex hardware to designing hardware based on software needs. These challenges are further compounded by the need to enable parallelism in mainstream workloads and application domains that have traditionally not had to employ parallelism.

Despite over four decades of research, few high-level parallel programming models are available to domain experts who are not at the same time experts in parallelism. Fortunately, this situation is starting to change. Frameworks such as Map-Reduce [14] successfully exploit implicit parallelism on distributed systems and have also been extended to heterogeneous platforms such as GPU [17] and FPGA [26], but unfortunately have a restricted programming model. Other models, such as CUDA [21] and OpenCL [19], provide a restricted programming model to the users of GPU accelerators, but also expose a significant amount of hardware details. The StarSs model builds a task-level model using a pragma-based approach (similar to OpenMP) to ease the burden of task-level programming for different architectures; its instantiations include Cell Superscalar [6] for the Cell broadband engine and GPUSs [5] for a system with multiple accelerators. Other works propose different approaches, such as extending a high-level language like Java to create domain-specific languages for tackling the problem of running on heterogeneous hardware. The DeLite project from Stanford [9] is an example of such a project. The Lime language from IBM Research [4] developed a compiler for source-to-source translation from a Java-like language to C, OpenCL and Verilog, and a runtime that can interact with the native libraries generated. In this paper, we propose a programming model that can express arbitrary task graphs, which none of the mentioned systems can do. Further, our research is orthogonal to these systems, because the individual tasks in our data-flow programming model can be still expressed in almost any language (including explicitly parallel languages), providing flexibility and reuse of the existing algorithms.

Concurrent Collections (CnC) is a macro data-flow model developed by Intel [18] for execution of C++ programs on homogeneous multicore processors. CnC is a general programming model

LCTES 2012, June 12-13, 2012, Beijing, China
Copyright © 2012 ACM 978-1-4503-1212-7. . . $10.00

[1] e.g. Tianhe-1A, the fastest supercomputer in top-500 Nov.2010 edition

that has some very desirable properties, such as determinism, data-race freedom and live-lock freedom [10]. The extensions we propose enable the use of accelerators by adding support for defining device steps in the language specification. This allows steps to run on multiple locations, which can include CPUs, GPUs and FPGAs. We demonstrate that these extensions, together with corresponding compiler and runtime improvements, result in a very significant performance improvement while maintaining a low energy envelope on a set of medical imaging applications and retaining the ease of programming championed by CnC.

This effort is the first step in achieving our long-term goal of learning from software when developing the appropriate hardware architecture and vice-versa — more precisely, through hardware-software co-design. Further, we detail how we achieve this first step: mapping an application onto a heterogeneous architecture with high performance and low energy consumption.

The contributions presented in this paper include

- Extending the CnC model with *tag functions* and *ranges* to enable automatic code generation of high-level operations for inter-task communication. This improves programmability and also makes the code more analyzable, opening the door for future optimizations.
- Introduction of task *affinity*, a tuning annotation in the specification language. Affinity is used by the runtime during scheduling and can be fine-tuned based on application needs to achieve better (faster, lower power, etc.) results.
- Introduction and development of a novel, data-driven runtime for the CnC model, developed in a C framework, using a research parallel programming model: Habanero-C (HC) [2] as a base language.
- Expanding the Habanero-C *dynamic work-stealing* runtime to allow cross-device stealing based on task affinity. Cross-device dynamic work-stealing is used to achieve load balancing across heterogeneous platforms for improved performance and is, to our knowledge, the first extension of this kind.
- A unique heterogeneous hardware platform test-bed that serves as a target for our software.
- Validation of the model's performance on the full software and hardware stack.

Section 2 presents our target heterogeneous platforms and our approach to abstracting hardware configuration. Sections 3 and 4 give an overview of the Concurrent Collection macro data-flow model, the enhancements for supporting hybrid execution, the interpretation of those enhancements to generate C code, and a description of the tool-chain. Section 5 describes Habanero-C, the lower-level parallel programming model on which we build our implementation, the extensions to the Habanero-C work-stealing runtime to enable cross-device scheduling and work-stealing, and the design of the Concurrent Collections runtime that combines all these enhancements to achieve better performance and/or lower power consumption. We present and discuss our experimental platform and results in section 6 and our conclusions in section 7.

2. Heterogeneous Platforms

2.1 Customizable Heterogeneous Platform (CHP)

Today's highly parallel general-purpose computing systems face serious challenges in terms of performance, power, heat dissipation, space, and cost. In domain-specific computing areas such as medical imaging, that require real-time performance, using accelerators such as GPUs or FPGAs can significantly boost the computing performance and throughput [11]. We envision that current and future computing systems will embrace customization and heterogeneity in order to match the requirements of domain-specific applications. Moreover, the integration of heterogeneous components (or accelerators) will become more extensive, because the integration can bring components that are currently far apart together to reduce the communication latency and boost the computation efficiency. Such trends already appear in the mobile segment where power consumption and form factors are critical issues.

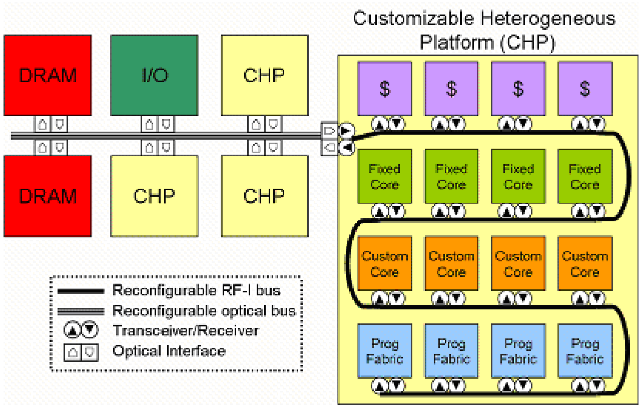


Figure 1. Customizable Heterogeneous Platform

Figure 1 shows a diagram of a Customizable Heterogeneous Platform or CHP [13], a futuristic heterogeneous platform. Specifically, a CHP includes 1) the integration of customizable cores and co-processors that will enable power-efficient performance tuned to the specific needs of an application domain and 2) reconfigurable high-bandwidth and low-latency on-chip and off-chip interconnects, which can be customized to specific applications or even specific phases of a given application.

There are several programming challenges inherent to the CHP. The first challenge is to find a way to efficiently program a particular heterogeneous component. This ultimately depends on the software kernel and the hardware architecture (e.g., ISA) of that component. Second, in the planning and design stage of the CHP, the co-design team will want to explore the design space and evaluate the same application on a set of different design configurations. The team would not want a significant code rewrite across different configurations. Upgrading from one CHP to another CHP with different amount of heterogeneous cores should not significantly affect the performance or require a code rewrite.

Manufacture of the fully customizable CHPs is still years ahead. In this paper, we first use a prototype machine consisting of off-the-shelf heterogeneous components (GPUs and FPGAs) to validate the concept. The programming environment presented in this paper allows us to switch easily from one configuration (e.g., CPU+GPU) to another (e.g. CPU+GPU+FPGA). This can be achieved simply by supplying a different platform description file. The implementation details are elaborated in the following sections.

2.2 Platform abstraction

The heterogeneous system has multiple heterogeneous components, and tasks can run on different components (likely using device-specific APIs). The platform abstraction file, at a minimum, has to describe the types of heterogeneous components and how many of each type the system contains. At this point, we are not yet modelling the interconnection topology of the CHP, which is a subject of our future work.

As an example, the XML code below describes a machine that has two CPU cores, one GPU, and one FPGA. This information is stored in an XML file following the Habanero Hierarchical Place Tree (HPT) format [25].

```
<?xml version="1.0"?>
<!DOCTYPE HPT SYSTEM "hpt.dtd">
<HPT version="0.1">
<place num="1" type="cpu" size="16G">
    <!-- the CPU global memory -->
<core num="2"/>
</place>
<place num="1" type="fpga" size="16G">
    <!-- the fpga global memory -->
 </place>
 <place num="1" type="nvgpu" size="4G">
     <!-- the GPU global memory -->
 </place>
 </HPT>
```

The platform description XML file lists the *places* available in the system. This information is further passed to the runtime system (Section 5) to determine the actual mapping of tasks to places. We plan to further enhance the description file to enable exposing more details of the target heterogeneous platform, in particular, the interconnection topology.

3. Programming Model

3.1 Concurrent Collections

Concurrent Collections [10] is a shared memory, dynamic, lightweight, task-based parallel programming model. A program in the CnC model is defined by a graph describing the dependences between serial pieces of computation called tasks. The model can also be described as a macro-dataflow coordination language, as it specifies how tasks interact with each other or depend on each other (data and control dependences).

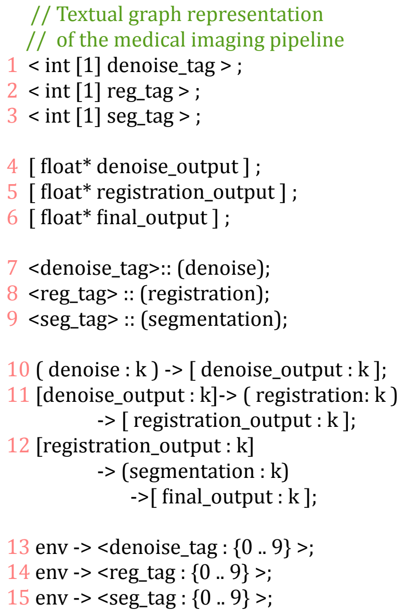
A graph specification for any program is built using three components: step, item, and control. These are grouped into collections:

1. A step collection is a group of tasks with the same functional behavior with respect to their inputs. They are declared in the text form using parentheses and are represented graphically using circles.
2. An item collection is a group of data items having the same type. They are textually represented with brackets and graphically represented with rectangles.
3. A control collection is a group of tags or keys used to create new steps within their respective collections, also known as tag collections. These are textually represented with angle brackets and are pictured as triangles.

The CnC specification also includes a special type of node, called the *environment*, which denotes the serial piece of code outside of the parallel CnC program, usually used to perform initial I/O, set up the CnC program execution, seed the CnC program with the input data, and read the results of the CnC computation.

Figure 2 shows an example of a textual CnC representation, while Figure 3 shows a graphical representation of the dynamic CnC graph for the same example (a medical imaging pipeline we shall later use in our results).

The Concurrent Collections programming model [10, 18] is designed to be implicitly parallel and easy to use by programmers with no knowledge of parallel programming. It is also more general than other deterministic programming models including dataflow and stream-processing and can incorporate static and dynamic forms of task, data, loop, pipeline, and tree parallelism.



```
 // Textual graph representation
// of the medical imaging pipeline
< int [1] denoise_tag > ;
< int [1] reg_tag > ;
< int [1] seg_tag > ;

[ float* denoise_output ] ;
[ float* registration_output ] ;
[ float* final_output ] ;

<denoise_tag>:: (denoise);
<reg_tag> :: (registration);
<seg_tag> :: (segmentation);

( denoise : k ) -> [ denoise_output : k ];
[denoise_output : k]-> ( registration: k )
          -> [ registration_output : k ];
[registration_output : k]
          -> (segmentation : k)
              ->[ final_output : k ];

env -> <denoise_tag : {0 .. 9} >;
env -> <reg_tag : {0 .. 9} >;
env -> <seg_tag : {0 .. 9} >;
```

Figure 2. Textual graph representation of the medical imaging pipeline

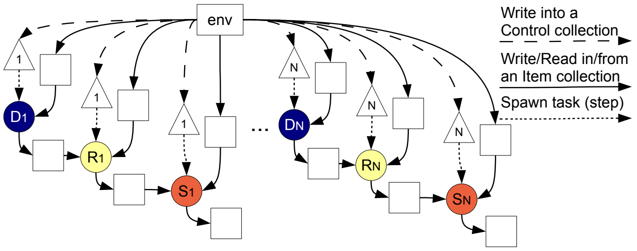


Figure 3. Dynamic graph representation for the medical imaging pipeline

The Concurrent Collection model uses the dynamic single-assignment rule for items added (*Put*) in an item collection, ensuring that the value that a task will read (*Get*) is the same every time. This property makes the model provably deterministic and race-free [10]. Live-lock may be possible if a task is waiting on an item that is never written and the implementation in the runtime involves a blocking operation. The CnC implementation we propose in this work is live-lock free. To differentiate our version of CnC from existing ones, we will refer to ours as CnC-HC.

The original definition and implementation of the CnC model was done by Intel, using C++ as the programming language to implement both the step code and the runtime. Our work uses Habanero-C ([2], section 5.1) for implementing CnC steps and for interaction between the CnC-HC runtime, CPU steps and device steps. This is the first publication to describe the CnC-HC system.

3.2 Extending the specification language

To use the CnC model in a manner that is easy to program, and enables the communication between CPUs and devices such as GPUs and FPGAs, we introduce a series of extensions to the model:

tag functions, ranges and task affinity. Tag functions and ranges are described in the following sections; we describe task affinity later in section 4 as the means to fine-tune the runtime scheduling of tasks on heterogeneous devices.

3.2.1 Tag functions

In order to achieve our first goal — making the model easy to program — we make two additions to the specification language: tag functions and ranges, both of which facilitate automatic code generation. We introduce tag functions in the CnC graph file specification to specify the relation between the tags of the steps and the tags of the data items that the step reads (a tag uniquely identifies a step or an item in a collection of data items). This new addition enabled us to create a tool that automatically generates code for reading the data items needed by a step (Section 3.3) and enables the runtime to use an efficient, data-driven implementation to schedule steps (Section 5). Instances of computation steps are uniquely identified by tags, which are tuples of symbolic names, e.g.,(x1, x2,...) . Thus, when writing a consumer and/or producer relation in the graph file, users can identify a step by a list of names they chosen for the components of the tag that prescribed the step. The items read and written can be specified by using a tag function f(x1, x2, ...). In Figure 2, lines 10-12 show the use of a simple identity tag function f(k)=k for the items read and written by the computation steps.

3.2.2 Ranges

Another addition to the CnC graph specification is the concept of ranges, which simplifies programming through code generation for reading groups of items, for writing groups of items, and for starting groups of steps. This extension targets a large number of applications, as many of them require that multiple pieces of data in a contiguous range be read or written or both. This is also in accord with constructs that involve split-join operations similar to map-reduce: a computation step writes N pieces of data that trigger N independent steps, each of which uses one item and gives a transformed piece of information; the results are then all read by a final step that computes the desired result. In the new model, this process can be accomplished by using ranges: the first step will write a range of data items and a range of control items; the latter will trigger the execution of as many steps as there are in the range; finally, the reduction step will read a range of items. In the example in Figure 2, on lines 13-15, we note that the environment will write a range of tags, thus starting a set of computation steps.

Ranges are specified in CnC using braces and two dots between the beginning and end of the range: { *start_range* .. *end_range* }.

3.3 Translator

All the code necessary for reading one or more items is auto-generated based on the information provided in the graph specification. Similarly, parallelism is ensured by auto-generated code and is transparent to the user. For this we developed a tool called *the CnC-HC translator*, which parses the CnC specification and translates it into Habanero-C code.

The translator uses the information provided by the user in the graph file to generate the necessary instructions for reading data to be used in a computation step. Parallelism is achieved by spawning each step as a separate task when all its input data is available, using the *async* primitive defined in Habanero-C. This is possible because in the model steps are functional with respect to their inputs. To handle synchronization issues, we created the CnC runtime, and, to achieve load balancing, we use the HC work-stealing runtime under the CnC runtime (both discussed in Section 5).

The translator also generates helper code for the steps. For example, if a step was defined to possibly create a control or data item, the step code will include a comment of how this would translate into code. The user can then use this code as he/she sees fit, though in most cases simply uncommenting the suggested code will give the desired user code.

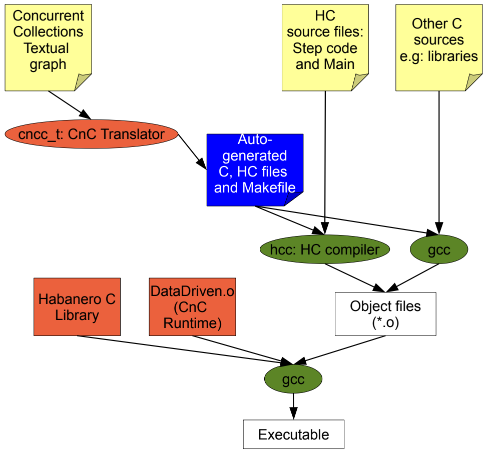

Concurrent Collections Textual graph
HC source files: Step code and Main
Other C sources e.g: libraries
cncc_t: CnC Translator
Auto-generated C, HC files and Makefile
hcc: HC compiler
gcc
Habanero C Library
DataDriven.o (CnC Runtime)
Object files (*.o)
gcc
Executable

Figure 4. CnC-HC Implementation Flow

Tag functions make the program flow more analyzable, and introducing ranges can extend CnC to other SIMD/SIMT architectures, such as GPUs, by allowing parallelism within a step. In addition to allowing domain experts to program without worrying about parallelism and the details of underlying C implementation of the model, this approach also opens the many opportunities for further research in automatic CnC program analysis and optimization.

3.4 Implementation flow

Figure 4 presents the implementation flow for a CnC-HC program.

A user has to take several steps to write a program using CnC-HC. First, they need to decompose the algorithm into steps and write a graph specification detailing the producer-consumer relationships between the computation steps. The CnC translator generates a series of "glue-code" files to enable transparent parallelism for the user, based on the guidelines provided in the graph file. It also creates code stubs as suggested and commented-out step code to handle the items written or steps enabled by a given step. To ensure an easy build, the translator also generates a makefile.

The user can then proceed to write the code for each of the computation steps and run the makefile to build the application. This uses the HC compiler and gcc compiler to generate an executable file. If additional libraries are required, they can be easily added in the provided makefile.

A common scenario in developing a program is making changes to the initial specification. In our case, if the user were to make changes to the graph file and rerun the translator, only the code linking the steps with the runtime will be overwritten to match the new dependences. Thus, if the computation had been written by a programmer prior to running the translator, their code will not be overwritten but preserved intact. If a step's definition were to change owing to changes in the specification (e.g., a step reading 3 items instead of 2), then the user will need to match his implementation with the new function prototype.

4. Tuning Annotations: Affinity

We have applied the CnC model in practice in the medical imaging domain, enabling the domain experts to reuse their previously implemented computational kernels. Certain kernels were only implemented on a particular platform, such as an FPGA. In contrast, other kernels had multiple implementations and could run on more than one platform (for example, on a CPU and a GPU, but not on an FPGA). From our preliminary experiments, we also knew that some steps would perform significantly better on a particular platform. Thus, we introduced the notion of *affinity* into the CnC graph specification to allow tuning of an application based on knowledge of where an algorithm can run and where it runs better if multiple variants are available. In this paper, we use the term "affinity" to indicate a scheduling priority rather than an indication of locality. Following are the lines of code from the graph in Figure 2 that we extended with an affinity annotation for each of the steps.

```
1   <denoise_tag >::(denoise @ CPU = 20, GPU=10);
2   <reg_tag> :: (registration @ GPU = 5, FPGA = 10);
3   <seg_tag> ::(segmentation @ GPU = 12);
```

Medical images are sent through the pipeline of image-processing kernels (denoising, registration and segmentation), resulting in an image that can be interpreted by a medical expert. All the steps in the above example are image-processing steps executing in sequence: the denoising step makes the image clearer, the registration step compares the image to previous scans, and finally the segmentation step localizes the desired area (in our case, a tumor or an aneurysm). Line 1 defines the denoising step, which can be run on both a CPU and a GPU but is more fitting for CPU execution. Line 2 defines the step executing registration, which can execute on a GPU and on an FPGA but is better suited for FPGA execution. Finally, line 3 defines the segmentation step, which only has a GPU implementation.

Currently, the values assigned as affinity are considered to prioritize the device on which a step should first attempt to execute; the relative quantitative differences are used to enable the "device worker" to use a small look-ahead and choose a task that needs to be executed sooner based on its higher priority. In this particular example, it may be desirable to execute the denoising steps with a higher priority in order to enable the registration step for that image. Similarly, we may want to execute the registration as soon as possible so that the segmentation can start. However, if we had a single GPU, note that the CPU and FPGA, respectively, can also execute the first two stages, while the last stage must run on the GPU. As a result, we have chosen the last stage to have the greater GPU affinity. We also notice that the first two stages run better on other platforms (CPU and FPGA, respectively), motivating a choice of affinities as defined by the example.

In general, a "tuning expert" would need to do the analysis we described above and choose appropriate affinity values. Thus, the affinity notion can be viewed as a tuning annotation [20], which we provide as a means of not only enabling the mapping of an application to a heterogeneous architecture but also tuning it according to a set of constraints (these can include throughput, energy efficiency or latency).

We are currently also exploring opportunities to define a quantitative notion of affinity in which we assume affinity values to be proportional relative to the expected performance on a given device (where a bigger number implies better performance). This feature would allow the runtime to perform even better informed scheduling decisions and possibly improve the results.

There has been previous work on tuning annotations in the CnC model [20], but they do not involve extensions for specifying affinities for heterogeneous architectures.

4.1 Theoretical bounds

The problem of obtaining theoretical bounds for schedules involving heterogeneous architectures is still an actively researched topic. Blumofe et al. investigated theoretical bounds for work-stealing among homogeneous tasks [7], while Agrawal et al. have shown that scheduling task flows with interleaved computation and communication on heterogeneous architectures is NP complete.[3].

5. Runtime Support

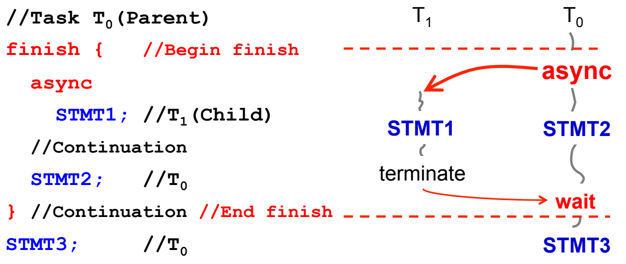
5.1 Habanero-C programming model

The Habanero-C (HC) language is a C-based task-parallel programming language developed at Rice University. In this section, we summarize key properties of HC and the Habanero model as described in [2, 13, 22] and then describe our extensions to the HC runtime system in Sections 5.2 and 5.4.

The two main features of HC that are relevant to this paper are

1. The *async* and *finish* constructs, which define lightweight dynamic task creation and termination and were originally defined in the X10 language [12].
2. Hierarchical place trees for locality control, which were originally defined for Habanero-Java [25].

The statement "**async** ⟨*stmt*⟩" causes the parent task to create a new child task to execute ⟨*stmt*⟩ *asynchronously* (*i.e.,* before, after, or in parallel) with the remainder of the parent task. Figure 5 illustrates this concept by showing a code schema in which the parent task, T_0, uses an `async` construct to create a child task T_1. Thus, `STMT1` in task T_1 can potentially execute in parallel with `STMT2` in task T_0.



```
//Task T0(Parent)
finish {    //Begin finish
  async
    STMT1;  //T1(Child)
  //Continuation
  STMT2;    //T0
} //Continuation //End finish
STMT3;      //T0
```

Figure 5. An example code schema with `async` and `finish`

`async` is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including statement blocks, for-loop iterations, and function calls. The project described in this paper, uses the `async` statement to create dynamic instances of CnC steps.

`finish` is a generalized join operation. The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all `async` tasks created within ⟨*stmt*⟩ have completed, including transitively spawned tasks. Each dynamic instance T_A of an `async` task has a unique *Immedia Enclosing Finish* (IEF) instance F of a `finish` statement during program execution, where F is the innermost `finish` containing T_A [24]. There is an implicit `finish` scope surrounding the body of `main()`, so program execution will only end after all `async` tasks have completed.

For example, the `finish` statement in Figure 5 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before T_0 executes `STMT3`. If T_1 created a child `async` task, T_2 (a "grandchild" of T_0), T_0 will wait for both T_1 and T_2 to complete in the `finish` scope before executing `STMT3`. As described in Section 5.4, we only use one level of `finish` when executing a CnC program because we include additional *data-driven* execution constraints on `async` tasks that control when `async` tasks become ready for execution.

Habanero-C allows arguments to be passed in an async statement in the following forms:

- IN(list of local variables): each of the values in the list is passed by value to the newly created task,
- OUT(list of local variables): each of the values in the list will override the local value with the value from the newly created (child) task once this finishes, or
- INOUT(list of local variables): each of the values in the list will be passed by value to the new task and, if modified inside this new task, the new value will be copied back to the parent task when the child finishes its execution.

The above clauses result in *shallow copies* of data structures.

The Habanero-C runtime uses a work-stealing scheduler that supports work-first and help-first policies [16]. In the work-first policy, the current thread will start to execute the newly spawned task, adding the continuation to its work queue from which other threads can steal it. In the help-first policy, the worker will make the child task available for stealing and continue executing the parent task. The work-first policy is good for scenarios when work-stealing is rare; however, it performs poorly in situations when the task continuation encompasses high parallelism that is not split into tasks, such as a for loop creating tasks using the async statement. It also has a provable memory bound relative to 1-processor execution, but it may overflow the stack in cases where the help-first policy does not. The help-first policy performs well when stealing is frequent; it uses little stack, but the memory usage can be large. Previous work [15] has shown how the two policies perform in different scenarios and that an adaptive approach that switches between the two policies can yield good performance improvement and efficient stack and memory usage.

For locality/affinity control, Habanero-C uses Hierarchical Place Trees (HPTs) [25]. HPTs define a hierarchical structure of computational nodes in the system, which are an abstraction of the underlying hardware. The nodes in the tree (places) can be CPU cores, groups of cores sharing different levels of cache, or devices such as GPUs or FPGAs. The Habanero-C HPTs are defined in an XML file and are also used as the Platform Abstraction for the underlying hardware as described in Section 2.2. For locality/affinity control, an optional `at` clause can be specified for a Habanero-C `async` statement of the form, "`async at(`*place-expr*`) ...`", where *place-expr* evaluates to a node in the HPT. This clause dictates that the child `async` task will be placed in the work queue at the specified place. Locality can be controlled by assigning two tasks with the same data affinity to execute in the same place. If the `at` clause is omitted, then the child task is scheduled by default to execute at the same place as its parent task. However, next section describes our extensions to the HC runtime system that allow for reassigning tasks across places if the initial assignment is unbalanced.

5.2 Dynamic work stealing

The original Habanero-C runtime system was implemented for homogeneous multicore processors, with support for work stealing among CPU workers. We extended the HC runtime to use the place/affinity annotation to facilitate task scheduling across heterogeneous processors by enabling work-stealing *among devices*. An *affinity* annotation is used to specify which variants of code are available for a given task (currently, these can be some subset of CPU, GPU, and FPGA).

Figure 6 shows a work-stealing scenario for a group of tasks for the graph presented in Figure 3. The environment first launches all the tasks $(D_1, R_1, S_1, D_2, R_2, S_2, \ldots)$ on devices specified by their initial affinities. In this particular example, denoising $(D_1, D_2, \ldots)$ tasks can run on a GPU device or on a CPU devic, and are initially launched on a GPU device. Segmentation $(S_1, S_2, \ldots)$ tasks can run on a GPU only. Registration $(R_1, R_2, \ldots)$ tasks can run on an FPGA or a GPU device and are initially launched on the FPGA device. The CPUs can steal denoising tasks from the GPU, and the GPU can steal registration tasks from the FPGA. Figure 8b shows a trace of an actual dynamic execution schedule for a set of 10 denoising, 10 registration and 10 segmentation tasks.

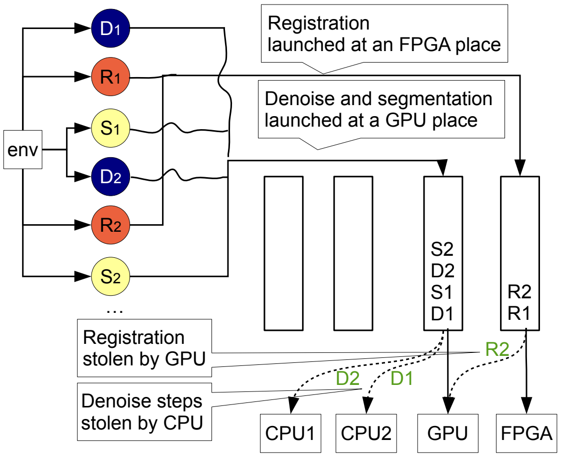

env
D1
R1
S1
D2
R2
S2
...
Registration launched at an FPGA place
Denoise and segmentation launched at a GPU place
S2
D2
S1
D1
R2
R1
Registration stolen by GPU
Denoise steps stolen by CPU
R2
D2
D1
CPU1
CPU2
GPU
FPGA

Figure 6. Dynamic work-stealing between devices for the medical imaging pipeline

In our implementation of work-stealing between devices, we also consider task affinity. Instead of stealing the first task available on a worker's deque, the thief worker will choose the one with the highest affinity from the first N tasks on the dequeue, where N is a small number typically less than 10 defined by the runtime (in our results we used a lookahead of N=5). For example, if the top 5 tasks in the FPGA queue have GPU affinities of 10, 20, 30, 15, and 5, the GPU worker, when stealing from the FPGA queue, will steal the one with the GPU affinity 30.

A CPU worker can steal from any kind of device, and any device can steal tasks from other devices (as long as the task can run on the thief's device). For performance reasons, we have disabled stealing from CPU queues by device workers. The reasons for this are twofold: 1) the CPU tasks usually have small granularity, and the overhead of launching a task on a GPU or FPGA device can be significant (see Section 6 for more details) and 2) allowing device workers to steal from CPU tasks would require locking the CPU queues in order to implement the N task lookahead, which would introduce significant overhead to CPU-to-CPU work stealing, currently implemented using a non-blocking algorithm [8].

Because of this restriction — disallowing device workers to steal tasks from a CPU worker — a task that has both CPU and some other device affinity needs to be launched on a device. In Habanero-C, the launch mechanism implies adding the task to the waiting queue for the device, not necessarily running it immediately. Thus, such a task may later be stolen by a CPU thread, whereas if it were launched on the CPU it would never be stolen by a device worker.

We combined the notion of affinity as described in the graph specification and the notion of device preference. The CnC translator generates appropriate HC code that passes the preferences specified by the user in the graph file to the work-stealing runtime.

Since steps can potentially run on different components, we used the Habanero-C library function *current_place()*, which can be used to determine on what type of hardware component the step runs. Based on the device type, the step code calls the appropriate code or library routine that is compiled for that device type.

5.3 Generalizing dynamic work-stealing

We looked into lifting the restriction of disallowing stealing from the CPU, while taking into account the granularity problem.

The solution we propose for CPU workers is splitting tasks that can run on the CPU into two categories: those that can run only on the CPU, which are viewed as small tasks, with fine granularity stored in a deque and those that can also execute on an accelerator, which are assumed to have larger granularity and which are stored in a queue. For the first category of tasks we use the same non-blocking algorithm such that a worker can pop from it's own deque or steal form other worker's deques using light-weight synchronization. For the second category, we use a blocking approach for popping or stealing from the CPU queues. With this approach, the tasks that will be the least likely to be stolen by the CPU are those tasks which were added to a device worker's queue (but can also be run by the CPU) as these tasks have larger affinity with some device. A CPU could, however, still steal from these in the event that no other tasks are available.

For device workers, we keep the same scheduling policy as before, which is a work-stealing algorithm among the device queues using a blocking approach and a lookahead of N tasks. We enhance this with stealing from CPU queues as well, but allowing tasks which have been added to device queues to have priority because of the affinity metric which made the initial decision of enqueuing the task for a device.

With the enhancements we propose, we allow the affinity metric to be used more accurately, thus defining a schedule more closely matching the indication of device preference.

5.4 CnC runtime

The CnC-HC model was developed on top of the Habanero-C (HC) programming language, and it uses the async and finish constructs available in HC. Since not all dependency graphs from CnC can be implemented using only async and finish constructs, we needed to extend the HC runtime with additional synchronization to support the CnC model. The previous implementations included techniques such as "rollback-and-replay", feasible for CPU-only scheduling, but impractical for use across heterogeneous processors. For this work, we took a new data-driven approach that differs from previous implementations of the CnC model [10], as it minimizes the number of spawns, thus reducing overhead.

In our implementation, the CnC runtime scheduler uses the tag functions to determine exactly which data items the step will get, and checks the availability of all those items. If not, it enqueues the step into a queue of tasks waiting for the particular (still unavailable) data item. When a *Put()* happens for that data item, the runtime iterates through all the steps that were waiting on that item and reevaluates whether they have all their data available. If so, the runtime launches the step using the HC async statement; otherwise, it continues putting the steps into queues of data items on which they are waiting.

In this section, we presented two runtime contributions: extending the HC runtime with cross-device work-stealing in order to achieve load balancing across heterogeneous processors and a new, data-driven runtime for CnC, which removes unnecessary spawns to reduce overhead.

6. Experimental Results

6.1 Prototyping platform

To validate our ideas on CHP, we built a single-node heterogeneous system integrating off-the-shelf components, including a multi-core CPU, many-core GPU, and FPGAs. We use the Convey HC-1ex [1] as our baseline platform. The form-factor of the platform is a 2-U rack-mountable server box. The motherboard has two PCI-e X16 slots, but no physical space to host a double-width GPU (e.g., GTX280) or Tesla compute card owing to form-factor issues. Currently we use a PCI-express expansion box to host a Tesla C1060. Figure 7 shows the structures of the coprocessor hardware of the Convey HC-1ex. The HC-1ex uses 4 Xilinx Virtex6 LX760 as the user FPGAs. The CPU and different FPGAs access the off-chip memory using a shared memory model. The system employs an on-board crossbar for the interconnection. Cache coherence is handled through the FSB protocols. Each FPGA has 16 external memory access channels. Eight physical memory ports are connected to eight memory controllers, which run at 300MHZ. The core design runs at 150MHZ. Thus, effectively, the design on each FPGA is presented with 16 "logical" memory access channels through time multiplexing. The Convey HC-1ex provides a very large bandwidth (80GB/s peak) and 16GB capacity for coprocessor side memory. In practice, we observe that around 30% to 40% of the peak bandwidth can be easily obtained. The FPGA-side off-chip memory system is designed to better support interleaved (short) data access rather than traditional cache-line burst access.

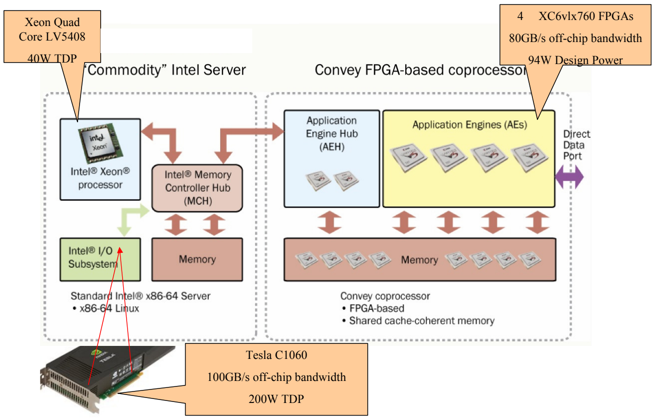

Xeon Quad
Core LV5408
40W TDP
"Commodity" Intel Server
Convey FPGA-based coprocessor
4 XC6vlx760 FPGAs
80GB/s off-chip bandwidth
94W Design Power
Intel® Xeon® processor
Intel® Memory Controller Hub (MCH)
Application Engine Hub (AEH)
Application Engines (AEs)
Direct Data Port
Intel® I/O Subsystem
Memory
Memory
Standard Intel® x86-64 Server
• x86-64 Linux
Convey coprocessor
• FPGA-based
• Shared cache-coherent memory
Tesla C1060
100GB/s off-chip bandwidth
200W TDP

Figure 7. Diagram of the Convey HC-1ex Hybrid Computer

Because the CPU and FPGAs share a common virtual memory space, while the GPU has its own memory space, we need to use device specific API to perform memory copy (e.g., *cudaMemcpy*).

Figure 7 also lists the TDP (or design power for FPGA) of the components. We use those numbers to estimate the energy consumption required by the computation. Actual system power will be higher, as the memory chip, chipset etc. also contributes to power consumption. Off-chip memory bandwidth for different components are shown as well. For all results reported in this section, we used the HPT file given in Section 2.2 as the the platform description for the CnC-HC runtime system.

6.2 Benefits of heterogeneous computing

We chose medical imaging as one of our primary application domain, as it has become a routine tool in the diagnosis and treatment of most medical problems.

Using accelerators can significantly speed up computationally intensive applications, such as medical imaging [11]. Table 1 shows the individual performance of the different application steps on CPUs, GPUs, and FPGAs. Note that the time measured is for computation kernels and excludes file I/O (around 2s overhead for each invocation).

We can see that GPUs and FPGAs deliver significant speedups compared to single-threaded CPU implementation. We also notice that different kernels prefer different accelerators. For the *registra-*

Table 1. Performance of Medical Imaging kernels on CPU, GPU and FPGA

	Denoise	Registration	Segmentation
Num. of Iterations	3	100	50
CPU	3.3s	457.8s	36.76s
GPU	0.085s (38.3×)	20.26s (22.6×)	1.263s (29.1×)
FPGAs	0.190s (17.2×)	17.52s (26.1×)	4.173s (8.8×)

tion kernel, FPGA delivers a higher speedup than GPU, with the roles reversed for *segmentation*.

Note that while it is possible to run multiple kernels on the FPGA, in practice that may involve a large reconfiguration overhead. In our experiments, we configured the FPGA to accelerate the *registration* kernel only.

6.3 Image Pipeline Example

We aim not only to evaluate a representative medical imaging pipeline that includes image denoising, image registration, and image segmentation kernels [13], but also observe how well our CnC-based mapping approach performs for such a pipeline. The algorithms that compose the medical image pipeline are stand-alone, complex algorithms that can be composed a variety of applications depending on the patient study needs. Apart from the example we outline in this paper, we mention studies that include multiple instances of the registration algorithm, used to characterize a patient's evolution over time using multiple clinical studies.

We first construct a CnC graph for the application:

```
< int [1] denoise_tag > ;
< int [1] reg_tag > ;
< int [1] seg_tag > ;

[ float* denoise_output ] ;
[ float* registration_output ] ;
[ float* final_output ] ;

<denoise_tag>:: (denoise@CPU = 20,GPU=10);
<reg_tag> :: (registration@GPU = 5, FPGA = 10);
<seg_tag> :: (segmentation@GPU = 12);

( denoise : k ) -> [ denoise_output : k ];
[denoise_output : k]-> ( registration: k )
                   -> [ registration_output : k ];
[registration_output : k] -> (segmentation : k)
                          ->[ final_output : k ];
```

Optionally, one can specify the control or data generated by the environment (the main thread). For example, adding

```
env -> <denoise_tag : {0 .. 9} >;
env -> <reg_tag : {0 .. 9} >;
env -> <seg_tag : {0 .. 9} >;
```

to the CnC specification indicates that we want to create an application that performs batch processing of the image pipeline, which processes 10 images. 0..9 are the *ranges* of the control tags to prescribe the computation steps.

In this CnC specification, we describe the list of computation tasks *denoise*, *registration*, and *segmentation*, and the input and output dependencies of each task. The CnC translator converts that description into a collection of Habanero-C files. Users can further edit those files to create a working implementation. For example, for the above-mentioned CnC file, the auto-generated skeleton for *registration.hc* is

```
 #include "Common.h"
void registration( int k, float* denoise_output0,\
Context* context){
   /*
   float* registration_output1;
   // allocate memory if necessary and fill
   //in values to put here
   char* tagregistration_output1=createTag(1, k);
   Put(registration_output1, \
           tagregistration_output1, \
    context->registration_output);
  */
}
```

The auto-generated code provides hints for the actual implementation, which follows:

```
#include "Common.h"
void registration( int k, float* denoise_output0,\
Context* context){

  float* registration_output1;
  if(current_place() == MEM_PLACE)
  {
registration_output1=REG_cpu(k,denoise_output0);
  }
  else if(current_place() == NVGPU_PLACE)
  {
registration_output1=REG_gpu(k,denoise_output0);
  }
  else if(current_place() == FPGA_PLACE)
  {
registration_output1=REG_fpga(k,denoise_output0);
  }
  char* tagregistration_output1 = createTag(1, k);
  Put(registration_output1,     \
        tagregistration_output1,\
context->registration_output);
}
```

From the code above, we can see that a function *current_place()* is used to obtain the current place of the task. Based on the type of the device, the step calls different routines. *MEM_PLACE* denotes a CPU device, *NVGPU_PLACE* denotes a GPU device, and *FPGA_PLACE* denotes an FPGA device. The scheduling and dependency checking is auto-generated and transparent to the user.

In the *main* function of the program, we simply initialize the required data structure (CnC graph) and create the computation *steps*. In this example, the application performs a batch processing on 10 images, where we create 10 instances of *denoise*, *registration*, and *segmentation*.

6.4 Benefit of dynamic work stealing across heterogeneous components

Without our proposed framework, one may simply construct a static mapping: for example, using CPU for denoising, FPGA for registration, and GPU for segmentation, essentially using components that are best-suited for each individual kernel. However, static scheduling does not keep track of resource availability. In the simple image pipeline, the execution time of segmentation is smaller

Table 2. Execution times and active energy with dynamic work stealing for the medical imaging pipeline

	Exec time	Estimated Active Energy
CPU only	2286s	69.8KJ
GPU only	276s	54.8KJ
CPU+GPU	251s	49.4KJ
CPU+GPU+FPGA (dynamic binding)	129s	36.1KJ
CPU+GPU+FPGA (static binding)	193s	23.0KJ

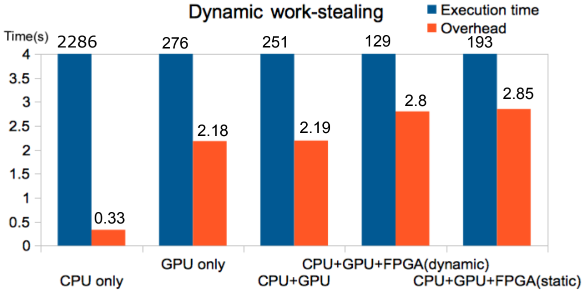


Figure 9. Execution time and overhead for the medical imaging pipeline

than registration, leaving the GPU idle for a long time when using a static schedule. Figure 8a shows the scheduling graph of the static scheme. Using a dynamic stealing achieves a better load balance and shorter overall execution time.

Table 2 shows the performance results for different hardware configurations: CPU only, GPU only, CPU + GPU, and CPU + GPU + FPGA (static or dynamic bindings).

We can see that with the cross-device stealing, a setup of CPU + GPU + FPGA with dynamic binding (Figure 8b) can perform better than the static scheme (Figure 8a). The energy column is computed by summing up the energy spent by each device that contributes to the computation, assuming 10W per core for the CPU, 200W for the GPU, and 94W for the FPGA, ignoring idle power. One of the reasons why the static binding has a lower energy estimate than dynamic binding in Table 2 is because idle power is ignored. If the idle power and the power of other system components are considered, we expect that dynamic binding will have a lower energy cost because of its shorter execution time. The data in Table 2 shows that different mapping policies may be needed for optimizing energy consumption or overall performance.

Note that in Figure 8, Dk means *denoise* instance k, Rk means *registration* instance k, and Sk means *segmentation* instance k. The graph in Figure 8b shows the effect of cross-device work-stealing. Initially 10 tasks of *denoise*($D0$ to $D9$) are pushed into the queue of GPU, of which $D4$ to $D9$ are stolen by the CPU. Similarly, the *registration* tasks are pushed into the queue of FPGA initially, but several task instances are stolen by GPU as well.

We want to point out that all results shown in Table 2 can be achieved by simply modifying the affinities of CnC description. For example, a static binding can be realized by

```
<denoise_tag>:: (denoise@CPU = 1);
<reg_tag> :: (registration@FPGA = 1);
<seg_tag> :: (segmentation@GPU = 1);
```

CPU-only, GPU-only and CPU+GPU can be constructed in a similar fashion.

Note that while the work-stealing runtime is quite powerful, the decisions it makes are simply based on the status of the queues and may actually hinder performance. For example, using

```
<reg_tag> :: (registration@CPU=1,GPU=2,FPGA=3);
```

to attempt to allow the CPU to help with running registration would slow down the overall performance, since even a single registration step takes a very long time to execute on a CPU. Nonetheless, the affinity annotations provide a powerful tuning capability to boost the overall application performance.

Examining the schedule resulting from dynamic binding in Figure 8b, we see room for a minor improvement if the scheduler chooses to give higher priority to first executing task $D3$ on a CPU worker, thereby enabling task $R3$ to start earlier on the GPU (assuming that tasks $D1$ and $D2$ also get scheduled on CPU workers). Exploring such improvements in scheduling algorithms is a subject for future work.

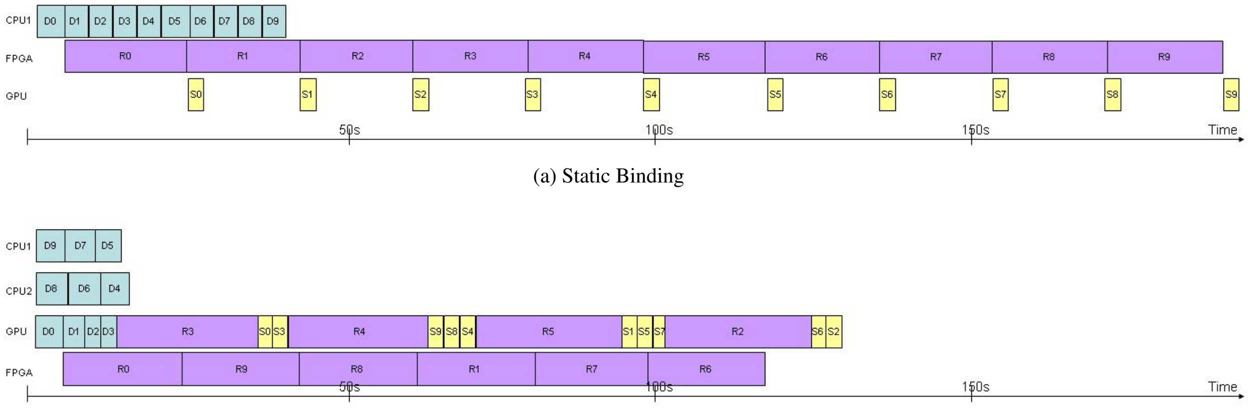
The graph in Figure 9 shows how the absolute performance compares to the overhead introduced by using the runtime. To measure the overhead, we ran the whole application without the actual work, replacing the computation with empty functions. We can see that in our current implementation, the hardware configuration with more heterogeneous components tends to have a larger overhead. However, these overheads are still quite small compared to the absolute overall execution time. Our future work will explore the opportunity to further reduce the data transfer overhead when the dependent task launches on the same heterogeneous component.

7. Conclusions and Future Work

In this paper, we showed how the Concurrent Collections (CnC) model can be mapped onto heterogeneous platforms in order to achieve high performance with low energy consumption, while preserving the ease of use of data-flow programming at a level appropriate for domain experts. This mapping relied on new extensions to the CnC model (tag functions, ranges, affinities) and new extensions to the Habanero-C (HC) runtime system (dynamic work-stealing across devices and data-driven execution of CnC tasks), all of which were described in the paper. We demonstrated the effectiveness of our proposed approach on a set of kernels taken from a real-world medical image-processing pipeline, that were mapped on to a unique prototype heterogeneous platform, which includes CPUs, GPUs and FPGAs.

We showed that our model offers efficient (0.52× of the power used by a CPU-only version) and competitive (17.72× speedup) results for the medical imaging domain, where getting a fast result, even when constrained by available power, is critical for facilitating diagnosis and treatment.

This work opens new research opportunities, which we plan to explore. First, the introduction of tag functions makes the representation of a computation in the CnC model more analyzable, offering the possibility to signal errors or problems in the specification through static analysis (such as defined steps not being started by any other step or the environment, items never being computed, or computed items never being used). In addition, the details introduced by the tag functions can help with scheduling choices and decisions on memory management. Secondly, we are exploring how the addition of ranges can further be used in defining data-parallel computations for SIMD/SIMT architectures like GPUs for fully automated data copying to and from a device. Thirdly, the ability to determine at runtime how the affinity values should be assigned, or have a profiling phase for getting accurate values for a particular platform, has the prospect of yielding even better performance, and we plan to look into it in more depth. Finally, our aim is to continue researching what primitives are useful for domain experts, in order to build a flexible and easy-to-program software for modeling



(a) Static Binding

(b) Dynamic Binding

Figure 8. CPU+GPU+FPGA Schedule Graph

their applications and the afferent customizable hardware platform on which it can be mapped with high performance and efficiency.

Acknowledgments

We thank the Center for Domain Specific Computing (NSF Expeditions in Computing Award CCF-0926127) that funded this work. Additional thanks to the Habanero team for their comments and feedback on this work.

References

[1] Convey HC-1ex, `http://www.conveycomputer.com/`.

[2] https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C.

[3] K. Agrawal, A. Benoit, and Y. Robert. Mapping linear workflows with computation/communication overlap. In *ICPADS '08*, pages 195–202, Washington, DC, USA, 2008. IEEE Computer Society.

[4] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA'10*, pages 89–108, 2010.

[5] E. Ayguadé et al. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par '09*, pages 851–862, 2009.

[6] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, 2006.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[8] R. D. Blumofe et al. CILK: An efficient multithreaded runtime system. *Proceedings of Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 207–216, July 1995.

[9] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *PACT '11: 20th International Conference on Parallel Architectures and Compilation Techniques, October 2011*.

[10] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent collections. *Scientific Programming*, 18:203–217, August 2010.

[11] A. Bui, K. Cheng, J. Cong, L. Vese, Y. Wang, B.Yuan, and Y. Zou. Platform Characterization for Domain-Specific Computing. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference*, ASPDAC '12, 2012.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, 2005.

[13] J. Cong, V. Sarkar, G. Reinman, and A. Bui. Customizable Domain-Specific Computing. *IEEE Design and Test*, (2:28):6–15, Mar 2011.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[15] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[16] Y. Guo et al. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS'09*, pages 1–12, 2009.

[17] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, 2008.

[18] Intel Corporation. Intel ® concurrent collections for C/C++. http://softwarecommunity.intel.com/articles/eng/3862.htm.

[19] Khronos OpenCL Working Group. The OpenCL Specification - Version 1.0. Technical report, The Khronos Group, 2009.

[20] K. Knobe and M. G. Burke. The Tuning Language for Concurrent Collections. In *16th Workshop on Compilers for Parallel Computing*, 2012.

[21] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[22] J. Payne, V. Cavé, R. Raman, M. Ricken, R. Cartwright, and V. Sarkar. DrHJ — a lightweight pedagogic IDE for Habanero Java. In *PPPJ'11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[23] V. Sarkar, W. Harrod, and A. E. Snavely. Software Challenges in Extreme Scale Systems. *SciDAC*, January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale.

[24] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08*, pages 277–288, New York, NY, USA, 2008. ACM.

[25] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2009.

[26] J. Yeung, C. Tsang, K. Tsoi, B. Kwan, C. Cheung, A. Chan, and P. Leong. Map-Reduce as a programming model for custom computing machines. In *Proceedings of the 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '08, pages 149 –159, Apr. 2008.