# Modeling and Mapping for Customizable Domain-Specific Computing

Zoran Budimlić[†] Alex Bui[‡] Jason Cong[‡] Glenn Reinman[‡] Vivek Sarkar[†]

[†]Rice University [‡]University of California, Los Angeles

## Abstract

In this article, we introduce the ongoing research in modeling and mapping for heterogeneous, customizable, parallel systems, as part of the effort in the newly established Center for Domain-Specific Computing (CDSC). This research combines the diverse backgrounds from multiple disciplines, including computer science and engineering, electrical engineering, medicine, and applied mathematics. The goal of this project is to look beyond parallelization and to focus on domain-specific customization as the next disruptive technology to bring orders-of-magnitude power-performance efficiency improvement to important application domains.

The project initially focuses on medical imaging applications, which provide an important tool in diagnosis and treatment of most medical problems, but many advances in this field have been constrained to the research environment due to a lack of computational power. Orders of magnitude in power-performance efficiency improvement that this project is expected to deliver will have a tremendeous impact on the applicability of the current applications and on the development of new ones.

***Keywords*** programming models, parallel programming, domain specific languages, declarative programming, data flow langauges, single assignment languages

## 1. Introduction

To meet ever-increasing computing needs and overcome power density limitations, the computing community has halted simple processor frequency scaling and entered the era of parallelization, with tens to hundreds of computing cores integrated in a single processor, and hundreds to thousands of computing servers in a warehouse-scale data center. Such highly parallel, general-purpose computing systems, however, still face serious challenges in terms of performance, power, heat dissipation, space, and cost. We believe the customizable domain-specific computing is a promising approach based on the following three observations:

1. Each user or enterprise typically has a high computing demand in one or a few selected application domains (e.g., graphics for game developers, circuit simulation for integrated circuit design houses, financial analytics for investment banks), while its other computing needs (e.g., email, word processing, web browsing) can be easily satisfied using existing computing technologies. Therefore, it is possible to develop a customizable computing platform where computing engines and interconnects can be specialized to a particular application domain, gaining significant improvements in power-performance efficiency as compared to a general-purpose architecture.
2. The performance gap between a totally customized solution (using an application-specific integrated circuit (ASIC)) and a general-purpose solution can be very large – can be several order of magnitude (e.g. see the case study in [21]).
3. However, it is extremely costly and impractical to implement each application in ASIC — the non-recurring engineering cost of an ASIC design at the current 45nm CMOS technology is over $50M [2] and the design cycle can easily exceed a year. There is a strong need for a novel architecture platform that can be efficiently customized to a wide range of applications in a domain or a set of domains to bridge the huge performance/power gap between ASICs and general-purpose processors.

Given these observations, the objectives of the CDSC are to develop a general (and reusable) methodology for creating novel and highly efficient customizable architecture platforms, and the associated compilation tools and runtime management environment to support domain-specific computing.

The basic concept of customizable architecture was introduced in the 1960s by Gerald Estrin [12]. Early successes in customizable computing were demonstrated in the 1990s, where certain compute-intensive kernels were



*SPLASH '10* October, Reno, NV.

manually mapped to FPGAs for acceleration, achieving significant speedup. Examples include the DECPeRLe-1 system [24], the GARP project [14], and commercial efforts by Cray and SRC Computers [1] (more examples of related work can be found in various FCCM Proceedings and a recent survey [13]). But these efforts faced several limitations, such as: the communication bottleneck between the host CPU and FPGAs; the fact that the customization was limited to FPGAs with little or no integration of the latest multicore architectures; the lack of high-performance reconfigurable interconnect structures; scalability limitations; and a restricted programming environment that often required manual coding in hardware design languages (HDL) or extensive rewriting of existing software for the FPGA implementation. Our research on domain-specific computing differs from the previous efforts in multiple ways. Our platform includes: 1) a wide range of customizable computing elements, from coarse-grain customizable cores to fine-grain field-programmable circuit fabrics; 2) customizable and scalable high-performance interconnects based on the RF-interconnect technologies; 3) highly automated compilation tools and runtime management systems to enable rapid development and deployment of domain-specific computing systems, and 4) a general, reusable methodology for replicating such success to different application domains. A significant challenge in this project is the modeling and mapping for heterogeneous, customizable, and parallel systems. In this paper, we shall present our inital progress in this area.

## 2. A Case Study: Medical Imaging Processing Domain

To demonstrate the customizable domain-specific computing technologies that will be developed in this project, we chose healthcare as our application domain, given its significant impact on the national economy and quality of life (e.g., 16% of the U.S. gross domestic product was spent on healthcare in 2005 [6]). In particular, we focus on the domain of medical imaging processing. Medical imaging is now a routine clinical tool in the diagnosis and treatment of most medical problems, but many advances in this field have been constrained to the research environment due to a lack of computational power. Several medical imaging algorithms are infeasible for real-time clinical use; and objective, automated quantitative methods that can enhance detection and evaluation are not widely used. Power and cost-efficient high-performance computation in this domain can have a significant impact on healthcare in terms of preventive medicine (e.g., virtual colonoscopy for colorectal cancer screening), diagnostic procedures (e.g., automatic quantification of tumor volume), and therapeutic procedures (e.g., pre-surgical decision-making and monitoring/analysis during surgery). Figure 1 shows a typical processing pipeline for medical imaging, with the following steps:


**Figure 1.** Medical Imaging Pipeline

- *Raw data acquisition:* Collecting the physical sensor data from the medical aparatus (e.g., x-rays, magnetic pulse sequences).
- *Image reconstruction:* Computes a series of images from physical sensor data.
- *Image restoration:* Removes noise and image artifacts (e.g., from environmental conditions, patient movement).
- *Registration:* Orients a given image to a reference image (e.g., of a healthy person, or an earlier image of the same individual).
- *Segmentation:* Identifies and extracts regions of interest in the image (e.g., a tumor).
- *Analysis:* Many kinds of feature analysis can be carried out in this step, such as measuring the size of a tumor, computing its growth rate (based on past segmentation results), etc.

Table 1 lists some typical algorithms used in these steps and their computation and communication patterns. Algorithms may vary considerably from one step to another, requiring different architecture support for the best efficiency. In our preliminary studies, we looked at the possibility of using GPUs and FPGAs for acceleration. For a bi-harmonic registration algorithm, GPU (Tesla C1060) provided 93x speedup while FPGA (Virtex-4 LX100) provided

| | Computation kernel | Communication scheme | Representative algorithm |
|---|---|---|---|
| **Reconstruction** | Dense and sparse linear algebra, optimization methods | Iterative; local or global communication | Compressive sensing |
| **Restoration** | Sparse linear algebra, structured grid, optimization methods | Non-iterative; highly parallel; local and global communication | Total variational algorithms |
| **Registration** | Dense linear algebra, optimization methods | Parallel, global communication | Optical flow/fluid registration |
| **Segmentation** | Dense linear algebra, spectral methods, MapReduce | Local communication | Level set methods |
| **Analysis** | Sparse linear algebra, n-body methods, graphical models | Local communication | Navier-Stokes equations |

**Table 1.** Algorithms in medical imaging pipeline

11x speedup (both measured against a Xenon 2GHz processor). However, for a 3D median denoising filter algorithm, GPU provided 70x speedup, while FPGA achieved close to 1000x speedup (due to bit-level parallel operations). It is clear that even in this rather narrow domain, no single homogeneous architecture can perform well on all these applications. This example underscores a need of customization an ability to adapt architectures to match the computation and communication requirements of an application.

## 3. Overall Approach

To realize the order-of-magnitude performance/power efficiency improvement via customization, yet still leverage economy of scale, we are developing a Customizable Heterogeneous Platform (CHP), consisting of a heterogeneous set of adaptive computational resources connected with high-bandwidth, low-power non-traditional reconfigurable interconnects.

Figure 2 illustrates our proposed CHP configuration with a set of fixed cores, customizable cores, programmable fabric, and a set of distributed cache banks ($). The design also includes the use of a high-performance reconfigurable on-chip and off-chip buses for high-bandwidth, low-latency communication between components.

***Customizable computing engines.*** Three component types that exhibit different levels of customization and parallelism are considered in CHP designs:

1. Fixed cores can vary dramatically in their energy efficiency, computational power, and area, but have limited reconfigurability: they can only make use of techniques like voltage or frequency scaling to adapt power/performance characteristics. An example of this kind of architecture is the IBM Cell, with a general-purpose PPE core and the more numerous, but simpler, SPE cores.
2. Customizable cores provide coarse-grain adaptation to application demand, offering a number of discrete, tunable options that can be set, with flexibility somewhere between FPGAs and fixed cores. It is possible to design cores with a rich set of tunable characteristics, such as register file sizes, cache sizes, datapath bit width, operating frequency, supply voltages, etc.
3. Programmable fabrics provide maximal flexibility, as they can be used to implement custom instructions and specialized co-processing engines to offload computation or accelerate core performance. They can implement customized circuits for complex operations in terms of the number of computing units, the types of computing units, the level of pipeline stages, etc. The preceding section illustrates the use of FPGAs for medical imaging acceleration, which was achieved with automatic C-to-FPGA compilation [11].

***Customizable interconnects.*** In addition to customizable computing engines, our CHP architecture will provide low-latency, high-bandwidth, and reconfigurable interconnects for data sharing between cores, co-processors, cache banks, and memory banks with the ability to accommodate the communication requirements of a particular application (or even different phases of the same application). We will consider adapting conventional interconnects to the demands of an application domain (e.g., using express-virtual channels [17]), or the use of novel interconnect technologies, such as RF interconnect (RF-I) [9]. Reconfiguration using RF-interconnects can be achieved by selectively allocating RF-I bandwidth between different components on-chip.

***Application modeling and software design.*** There is a natural tension between hardware-first and software-first approaches to building domain-specific systems. In the former, one can start with a representative workload of existing domain applications, and use their characteristics to create a customized CHP that, in turn, drives the design of a domain-specific language and compiler extensions. The hardware-first approach represents the usual practice of ”software playing second fiddle to hardware”, exemplified by recent multicore processors like the Cell where investigation of general-purpose programming models and compiler innovations only started after the hardware design was completed. In the software-first approach, one can use the workload characteristics to drive the definition of a domain-specific programming language and compiler extensions that, in turn,

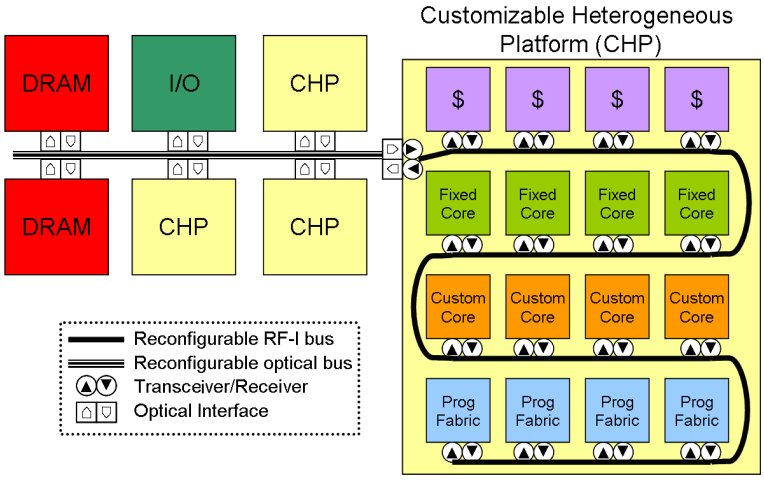


**Figure 2.** Platform for domain-specific computing

drive the creation of the CHP. The software-first approach has been pursued less often and has the danger of leading to special-purpose hardware with applicability to narrow application domains. As there are well-known limitations in both approaches, we instead use the following three-stage approach to carefully balance software and hardware considerations in the spirit of software-hardware co-design:

1. *Domain-specific modeling.* In the first stage, we create a representative set of executable application models using domain-specific language extensions (DSLEs) and a domain-specific coordination graph (DSCG) notation, both of which are designed to be accessible to domain (programming) experts. These models can be used to reveal inherent high-level properties of the application domain, such as intrinsic parallelism and communication topologies. This domain-specific modeling is used as an input to later CHP creation and CHP mapping stages.
2. *CHP creation.* In this second stage, we use the application models to design and implement an optimized set of hardware resources (CHP) for a particular application domain (or a set of related domains). The CHP creation determines how many cores, how much cache, which custom instructions, what amount of customization and reconfiguration, and what sorts of mapping transformations are useful for customization.
3. *CHP mapping.* Given a set of application models and an optimized CHP for a given domain, the third problem is CHP mapping, which develops domain-specific compilation and runtime systems that enable optimized mappings of the applications in the domain to the heterogeneous resources of a given CHP. This stage also determines what configuration and transformation settings should be selected for configurable CHP resources in different phases of a program.

This three stage approach is illustrated in Figure 3. We think that it is inefficient to apply the current general-purpose programming models for heterogeneous hardware, such as CUDA [15] and the Cell SDK, to CHP modeling and mapping. These existing models force the programmer to exploit customizable hardware at the lowest possible levels of hand-partitioned code and explicit data transfers that are tied to specific hardware structures. Such frameworks result in significant rewrites when the application needs to be re-partitioned for execution on different hardware configurations or platforms. These approaches also miss out on opportunities for hardware customization for different application phases. In the remainder of this paper, we shall discuss our ongoing effort for modeling and mapping for customizable, domain-specific platforms.

## 4. Domain-specific Modelling: the CnC Programming Model

In the modeling stage of the software deployment process on a CHP, we use the Concurrent Collections (CnC) programming model [7], which is built on past work on TStreams [16]. CnC belongs to the same family as dataflow and stream-processing languages—a program is a graph of serial kernels, communicating with one another. In CnC, those computations are called *steps*, and are related by control and data dependences. CnC is provably deterministic. This limits CnC's scope, but compared to its more narrow counterparts (StreamIT, NP-Click, etc), CnC is suited for many applications—incorporating static and dynamic forms of task, data, loop, pipeline, and tree parallelism.

The three main constructs in CnC are *step collections*, *data collections*, and *control collections*. These collections

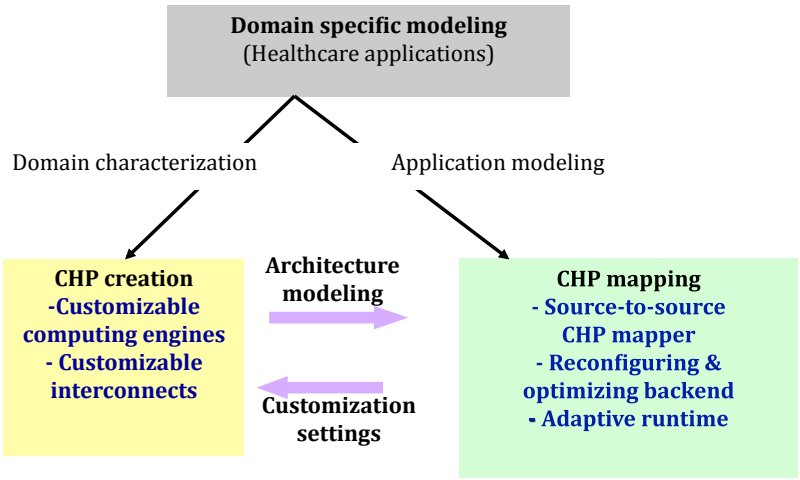


**Figure 3.** Illustration of the overall approach

and their relationships are defined statically. But for each static collection, a set of dynamic *instances* is generated at runtime.

A step collection corresponds to a specific computation (a procedure), and its instances correspond to invocations of that procedure with different inputs. A control collection is said to *prescribe* a step collection—adding an instance to the control collection will cause a corresponding step instance to eventually execute with that control instance as input. The invoked step may continue execution by adding instances to other control collections, and so on.

Steps also dynamically read and write data instances. If a step might touch data within a collection, then a (static) dependence exists between the step and data collections. The execution order of step instances is constrained only by their data and control dependencies. A complete CnC specification is a graph where the nodes can be either step, data, or control collections, and the edges represent *producer*, *consumer* and *prescription* dependencies. The following is an example snippet of a CnC specification (where bracket types distinguish the three types of collections):

```
// control relationship:
// myCtrl prescribes instances of step
<myCtrl> :: (myStep);
// consume from myData, produce to myCtrl, myData
[myData] → (myStep) → <myCtrl>, [myData];
```

For each step, like `myStep` above, the domain expert provides an implementation in a separate programming language and assembles the steps using a CnC specification. In the modeling stage of the CHP software deployment, the step implementation code can be Java, Python or Matlab. In the mapping stage, the step implementation code is in Habanero-C, which is described in Section 5. (In this sense CnC is a coordination language.) The domain expert says nothing about how operations are scheduled which depends on the target architecture. The tuning expert then maps the CnC specification to a specific target architecture, creating an efficient schedule. Thus the specification serves as an interface between the domain and tuning experts. This differs from the more common approach of embedding parallelism constructs within serial code.

A whole CnC program includes the specification, the step code, and the environment. Step code implements the computations within individual graph nodes, whereas the environment is the external user code that invokes and interacts with the CnC graph while it executes. The environment can produce data and control instances, and consume data instances.

Inside each collection, control, data, and step instances are all identified by a unique *tag*. These tags generally have meaning within the application. For example, they may be tuples of integers modeling an iteration space. They can also be points in non-grid spaces—nodes in a tree, in an irregular mesh, elements of a set, etc. In CnC, tags are arbitrary values that support an equality test and hash function. Each type of collection uses tags as follows:

- Putting a tag into a control collection will cause the corresponding steps (in prescribed step collections) to eventually execute. A control collection $C$ with tag $i$ is denoted $< C : i >$.
- Each step instance is a computation that takes a single tag (originating from the prescribing control collection) as an argument. The step instance of collection $(foo)$ at tag $i$ is denoted $(foo : i)$.
- A data collection is an associative container indexed by tags. The entry for a tag $i$, once written, cannot be overwritten (dynamic single assignment). The immutability of entries within a data collection is necessary for determinism. An instance in data collection $x$ with tag "$i, j$" is denoted $[x : i, j]$.

The colon notation above can also be used to specify *tag functions* in CnC. These are declarative contracts that constrain the data access patterns of steps. For example, a step indexed by an integer $i$ which promises to read data at $i$ and

produce $i+1$ would be written as "`[x: i]` → `(f: i)` → `[x: i+1]`".

Because tags are effectively synonymous with control instances we will use the terms interchangeably in the remainder of this paper. (We will also refer to data instances simply as *items*, and operations on collections as $puts$ and $gets$.)

### 4.1 Simple Example

The following simple example illustrates the task and data parallel capabilities of CnC. This application takes a set (or stream) of strings as input. Each string is split into words (separated by spaces). Each word then passes through a second phase of processing that, in this case, puts it in uppercase form.

```
<stringTags> :: (splitString);   // step 1
<wordTags>   :: (uppercase); // step 2
// The environment produces initial inputs
// and retrieves results:
env → <stringTags>, [inputs];
env ← [results];
// Here are the producer/consumer relations
// for both steps:
[inputs] → (splitString) → <wordTags>, [words];
[words]  → (uppercase) → [results];
```

The above text corresponds directly to the graph in Figure 4. Note that separate strings in `[inputs]` can be processed independently (data parallelism), and, further, the `(splitString)` and `(uppercase)` steps may operate simultaneously (task parallelism).

The only keyword in the CnC specification language is `env`, which refers to the *environment*—the world outside CnC, for example, other threads or processes written in a serial language. The strings passed into CnC from the environment are placed into `[inputs]` using any unique identifier as a tag. The elements of `[inputs]` may be provided in any order or in parallel. Each string, when split, produces an arbitrary number of words. These per-string outputs can be numbered 1 through $N$—a pair containing this number and the original string ID serves as a globally unique tag for all output words. Thus, in the specification we could annotate the collections with tag components indicating the pair structure of word tags: e.g. `(uppercase: stringID, wordNum)`.

The step implementations (user-written code for `splitString` and `uppercase` steps, ommitted here due to space constraints), specification file, and code for the environment together make up a complete CnC application. Current implementations of CnC vary as to whether the specification file is required, can be constructed graphically, or can be conveyed in the host language code itself through an API.

Figure 5 shows a CnC model for the Rician Denoising algorithm, which is a key component in the medical imaging pipeline on Table 1. The main components of the algorithm are modeled as CnC steps, and the data that is communicated between steps is modeled as CnC item collections. Depending on the platform, the steps can execute in parallel, and even on different hardware: some steps can execute on a GPU, some on a CPU, and some on an FPGA.

Figures 6 and 7 show how CnC can be used for modeling a part of the Fluid Registration algorithm, another key component in the medical imaging pipeline. Figure 6 shows an excerpt from a sequential task graph of the Fluid Registration algorithm. Figure 7 shows the CnC model for the same algorithm. The transformation only involved identification of some additional steps (convolution3D and process), capturing the data passed as arguments into item collections, and discovering the control dependencies and capturing those in control collections. The CnC model of the same algorithm now reveals a 6-way parallelism available in this code snippet alone (three instances of the convolution3D step can be run in parallel, and within each one of those, two fftw steps can be done in parallel.

## 5. CHP Mapping: the Habanero-C Language, Compiler and Runtime

The CHP mapping stage requires a flexible and portable solution to software deployment on a heterogeneous and customizeable CHP platform. In the mapping stage, we use the same CnC coordination graph from the modeling stage, with the step code implemented in Habanero-C, which is described below.

The Habanero Multicore Software Research project [3] adresses the multicore software challenge by developing new programming technologies — languages, compilers,runtimes, concurrency libraries, and tools — that support portable parallel abstractions for future multicore hardware with high productivity and high performance. Habanero-C is a language, compiler and runtime that integrates four orthogonal constructs with C language to support task parallelism: lightweight dynamic task creation and termination using *async* and *finish* constructs, locality control with task and data distributions using the *place* construct [8], mutual exclusion and isolation among tasks using the *isolated* construct [4], and collective and point-to-point synchronization and reduction using the "phasers" construct [22, 23].

1. Lightweight dynamic task creation and termination using *async* and *finish* constructs. The statement "*async [(place)] [phased(c...)] stmt* " creates a new child activity that executes statement *stmt*, registered on all phasers in the phased(...) list. An *async* statement can optionally include a *place* clause that serves as an affinity hint to constrain the execution of the *async* to a designated subset of workers. The statement '*finish stmt*' executes *stmt* and waits until all (transitively) spawned *async* tasks have terminated. The two constructs are similar to OpenMP *task* and *taskwait*, but more flexible.
2. Locality control with task and data distributions using the *place* construct [8]. A *place* in Habanero enables co-location of asynchronous tasks and shared mutable

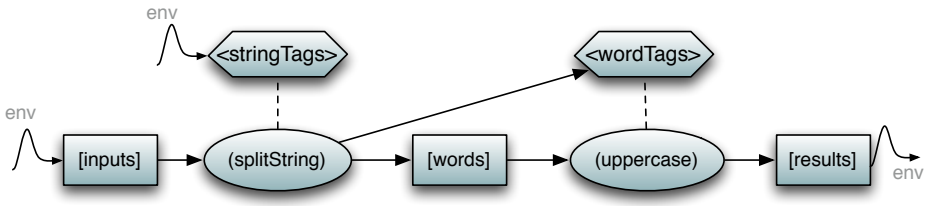


**Figure 4.** A CnC graph as described by a CnC specification. By convention, in the graphical notation specific shapes correspond to control, data, and step collections. Dotted edges represent prescription (control/step relations), and arrows represent production and consumption of data. Squiggly edges represent communication with the environment (the program outside of CnC)

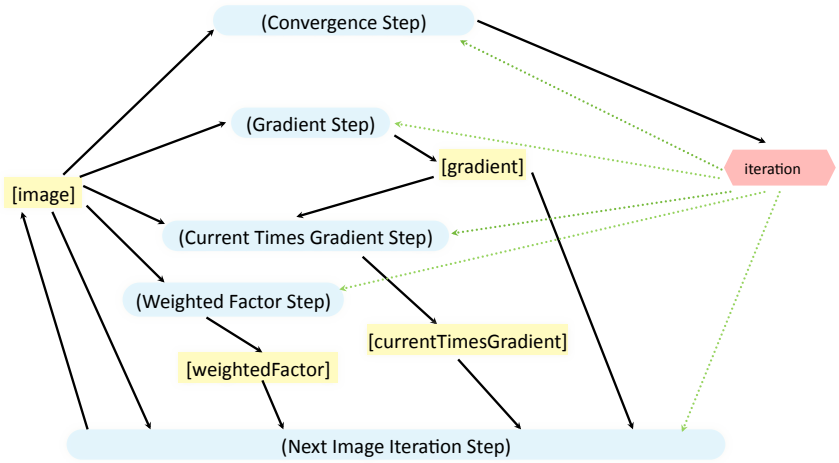


**Figure 5.** CnC Model for the Rician Denoising algorithm

locations. Habanero places are used to support both data distributions and computation distributions.

3. Mutual exclusion and isolation among tasks using the *isolated* construct [4]. The statement '*isolated [(place list)] stmt*' executes *stmt* in isolation with respect to the list of places. As advocated in [18], we use the *isolated* keyword instead of *atomic* (as it is named in X10) to make explicit the fact that the construct supports weak isolation rather than strong atomicity.
4. Habanero integrates collective and point-to-point synchronization, barriers, semaphores, and streaming computations into a single construct called *phasers* designed for dynamic asynchronous parallelism. Activities can use phasers to achieve collective barrier or point-to-point synchronization. The *next* statement suspends the activity until all phasers that it is registered with can advance. Phasers are dynamic (number of activities using a phaser can change at runtime), deadlock-free in absence of explicit wait operations, and lightweight. These properties distinguish phasers from synchronization constructs in the past including barriers [19], counting semaphores [20], and X10's clocks [10].

Habanero-C is translated by the compiler into C code with library calls to the runtime functions, that can be compiled by a native C compiler to the target processing unit, which can be a general-purpose CPU, a customized CPU, a GPU or an FPGA.

The current Habanero-C compiler and runtime implementation includes the *async, finish, phaser* and *place* constructs, while the support for *isolated* is under way.

The Habanero-C runtime implements work-stealing for lightweight tasks [5] for improved performance and scalability. A unified runtime for heterogeneous platforms that will support work-stealing across GPU threads is also under development.

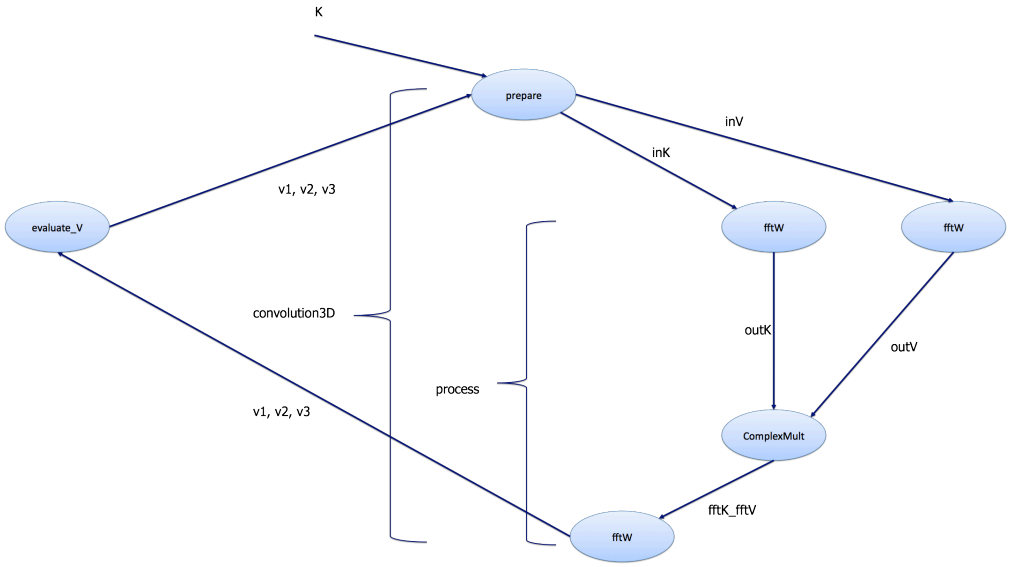


**Figure 6.** Sequential task graph for the Fluid Registration algorithm

## 6. Conclusions

This paper introduces the ongoing research in modeling and mapping for heterogeneous, customizable, parallel systems, as part of the effort in the newly established Center for Domain-Specific Computing (CDSC). This research combines the diverse backgrounds from multiple disciplines, including computer science and engineering, electrical engineering, medicine, and applied mathematics. The project looks beyond parallelization and focuses on domain-specific customization as the next disruptive technology to bring orders-of-magnitude power-performance efficiency improvement to important application domains.

We have presented a case study that focuses on medical imaging processing domain, which provide many important tools in diagnosis and treatment of most medical problems. We introduce a novel approach for domain-specific modeling using the Concurrent Collections programming model, which enables a high-level design approach that exposes the application's parallelism whle allowing the domain expert to focus on application design and not on the platform details. For mapping of the applications to the heterogeneous domain-specific hardware, we introduce Habanero-C language, compiler and runtime which provides a high-level parallel programming model designed for use by tuning experts on heterogeneous and highly parallel hardware.

Orders of magnitude in power-performance efficiency improvement that this project is expected to deliver will have a tremendeous impact on the applicability of the current applications and on the development of new ones.

## Acknowledgments

The Center for Domain-Specific Computing is funded by the NSF Expedition in Computing Award CCF-0926127. Other faculty members in CDSC are Denise Aberle, Richard Baraniuk, Frank Chang, Tim Cheng, Jens Palsberg, Miodrag Potkonjak, P. Sadayappan, and Luminita Vese. Their participation in this project is greatly appreciated. The authors would also like to thank William Hsu, Gene Auyeung and Igor Yanovsky at UCLA, and Alina Sbirlea, Dragos Sbirlea and Sagnak Tasirlar for their help in modeling the Fluid Registration algorithm using CnC.

## References

[1] http://www.srccomp.com/techpubs/techoverview.asp.

[2] International technology roadmap for semiconductors. http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[3] Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736, New York, NY, USA, 2009. ACM.

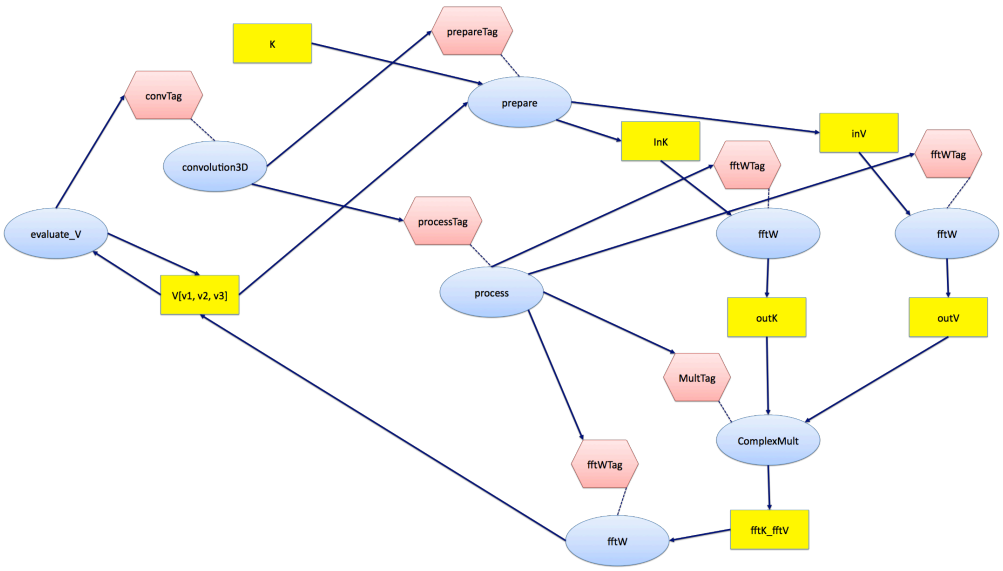


**Figure 7.** CnC graph for the Fluid Registration algorithm


[4] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT'09, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 41–52, Washington, DC, USA, Sep 2009. IEEE Computer Society.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work-stealing. In *Proceedins of the 35th Annual IEEE Conference on Foundations of Computer Science*, 1994.

[6] Christine Borger, Sheila Smith, Christopher Truffer, Sean Keehan, Andrea Sisko, John Poisal, and M. Kent Clemens. Health Spending Projections Through 2015: Changes On The Horizon. *Health Aff*, 25(2):w61–73, 2006.

[7] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. CnC programming model. *Journal of Scientific Programming (to appear)*, 2010.

[8] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 11–22, New York, NY, USA, 2008. ACM.

[9] M.F. Chang, J. Cong, A. Kaplan, M. Naik, G. Reinman, E. Socher, and S.-W. Tam. Cmp network-on-chip overlaid with multi-band rf-interconnect. pages 191 –202, feb. 2008.

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[11] J. Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Platform-based behavior-level and system-level synthesis. pages 199 –202, sep. 2006.

[12] Gerald Estrin. Organization of computer systems: the fixed plus variable structure computer. In *IRE-AIEE-ACM '60 (Western): Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, pages 33–40, New York, NY, USA, 1960. ACM.

[13] Scott Hauck and André DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*. Morgan Kaufmann, November 2007.

[14] J.R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. pages 12 –21, apr. 1997.

[15] W. M. Hwu. Performance insights on executing non-graphics applications on CUDA on the NVIDIA GeForce 8800 GTX, 2007. Invited talk at Hot Chips 19.

[16] Kathleen Knobe and Carl D. Offner. Tstreams: A model of parallel computation (preliminary report). Technical Report HPL-2004-78, HP Labs, 2004.

[17] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express virtual channels: towards the ideal interconnection

fabric. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 150–161, New York, NY, USA, 2007. ACM.

[18] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[19] OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org, 2006.

[20] V. Sarkar. Synchronization using counting semaphores. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 627–637, New York, NY, USA, 1988. ACM.

[21] Patrick Schaumont and Ingrid Verbauwhede. Domain-specific codesign for embedded security. *Computer*, 36:68–74, April 2003.

[22] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2008. ACM.

[23] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[24] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé H. Touati, and Philippe Boucard. Readings in hardware/software co-design. chapter Programmable active memories: reconfigurable systems come of age, pages 611–624. Kluwer Academic Publishers, Norwell, MA, USA, 2002.