LLNL-TR-457671

GETTING TO EXASCALE: APPLYING NOVEL PARALLEL PROGRAMMING MODELS TO LAB APPLICATIONS FOR THE NEXT GENERATION OF SUPERCOMPUTERS

E. Dube, L. Nau, C. Shereda, with contributions from L. Harris

September 29, 2010

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

TABLE OF CONTENTS

1. Executive Summary

As supercomputing moves toward exascale, node architectures will change significantly. CPU core counts on nodes will increase by an order of magnitude or more. Heterogeneous architectures will become more commonplace, with GPUs or FPGAs providing additional computational power.

Novel programming models may make better use of on-node parallelism in these new architectures than do current models. In this paper we examine several of these novel models – UPC, CUDA, and OpenCL –to determine their suitability to LLNL scientific application codes.

Our study consisted of several phases:

- We conducted interviews with code teams and selected two codes to port.
- We learned how to program in the new models and ported the codes.
- We debugged and tuned the ported applications.
- We measured results, and documented our findings.

We conclude that UPC is a challenge for porting code, Berkeley UPC is not very robust, and UPC is not suitable as a general alternative to OpenMP for a number of reasons. CUDA is well supported and robust but is a proprietary NVIDIA standard, while OpenCL is an open standard. Both are well suited to a specific set of application problems that can be run on GPUs, but some problems are not suited to GPUs. Further study of the landscape of novel models is recommended.

For readers who are:	**We recommend sections:**
Computation directorate management	2, 7, 9, 10
Systems administrators and non-DEG LC staff	2, 7-10
CAR/CASC researchers and managers	2-7, 9, 10
Application development teams	All
Development environment group staff	All

2. Introduction: Overview of This Study and Its Objectives

2.1. Background

It is widely expected that the computer systems anticipated in the 2015 – 2020 timeframe will be qualitatively different from current and past computer systems. They will be built using massive multi-core processors with hundreds of cores per chip. Their performance will be driven by parallelism, constrained by energy, and with all of their parts, they will be subject to frequent faults and failures. In this new generation of supercomputing, coined *Exascale Computing*, the number of nodes and the network will not dramatically change, but the system size and the node architecture are expected to shift radically. There will be multiple memory types, including programmable (scratchpad) memory along with generally more heterogeneous and hierarchical systems than today. The memory to FLOPS ratio is expected to worsen.

For exascale computing, the main programming environment challenges are expected to be within the new node rather than across nodes, since that is where the biggest changes will occur. The total number of nodes will not increase dramatically, so the current practice of using MPI between nodes to this scale provides one option of utilizing the exascale systems. In this hybrid model, OpenMP, UPC, Co-Array Fortran, or GPU-centric models such as CUDA or OpenCL can be used to achieve intranode parallelism. Another option is to utilize unified programming models at the global level (UPC, Co-Array Fortran, Chapel, X10, etc.)

The DOE has identified two 'swim lanes' for reaching exascale. The swim lanes define different architectural approaches. They are:

a) using nodes with hundreds of CPU cores, and making use of these many cores;
b) using nodes with GPU accelerators , and parallelizing applications by making use of these GPUs.

The ***Anticipated Exascale Timeline*** is listed below:

Year	Anticipated Exascale Timeline
2010-2011	develop abstract node/machine model
2010-2012	initial programming models development
2012-2013	early demonstration of programming models, generating course corrections
2013-2015	continued programming models development
2013-2015	application development in programming models
2015	deployment on 100 petaflop systems
2018	deployment on exaflop systems

With the short time frame, i.e., deployment on 100 petaflop systems by 2015, there is insufficient time to develop new programming models from scratch. The current plan is to evolve and extend existing programming models.

2.2. Need and Objectives

This study was conducted in response to the need of application teams to determine how to make effective use of the future exascale environment at Livermore. Making use of the forthcoming on-node parallelism will certainly require code modification and creative approaches, and may necessitate the adoption of new language paradigms. We investigated several of these new models. Our objectives were to determine:

a) the relative ease in acquiring the necessary programming skills;
b) the relative ease in porting code;
c) the robustness of the development environment and tools associated with the language model;
d) the performance of applications coded with the new model.

We selected UPC, CUDA, and OpenCL as the target language models for this study. UPC is a language that can either be used as an alternative to OpenMP for on-node parallelism, or for both intra- and internode parallelism. CUDA and OpenCL are language models for parallelizing code using GPUs . GPUs are one possible means for achieving high levels of intranode parallelism and reaching exascale.

In one portion of the study we compared the performance of an application ported to each of these target models against the same application parallelized with OpenMP. OpenMP is a relatively well-known and stable model for achieving intranode parallelism, so it provides a good baseline for comparison. The code is an intranode-only program. There is a routine which mimics an MPI data exchange within the code, so the target parallelism model is a hybrid one.

We also ported a separate laboratory application just to UPC and measured its performance. In this portion of the study we modeled the unified or holistic approach of using a single notation for both inter- and intra-node parallelism.

2.3. Our Approach

Before beginning, we selected the initial programming models to study based on resources, interest, and applicability. The study consisted of several phases, broadly:

- We began by conducting initial interviews with code teams to determine their needs and their understanding of the move toward exascale.
- We then selected our target applications.
- We learned how to program using the new models, learned the details of the target applications, and ported the code, documenting our findings along the way.
- We measured performance, collected results, and looked for opportunities to improve performance.
- We documented our findings.

3. Programming Paradigms Studied

3.1. UPC Overview

UPC, which stands for Unified Parallel C, is a parallel programming language that is an extension to the C language. UPC is a PGAS (Partitioned Global Address Space) language. This means that parallelism is achieved through the use of shared memory and work sharing across independent *threads* of execution.

3.1.1. Shared Memory

Shared memory variables in UPC form the foundation of UPC's parallelism. Rather than exchanging data across threads through explicit communication as in MPI, information is exchanged primarily through the use of shared memory.

Shared memory variables are declared through the use of the *shared* qualifier. In UPC, shared variables are always of global scope and must be declared globally; there is no provision for local shared variable declarations. This limitation is discussed further in Section 6.1.2.

If a shared variable is a scalar variable, it will be allocated by the first thread of a job (thread 0). If it is a static array, it will be allocated according to a qualifier called the *blocking factor*. If it is a dynamic shared array, it will be allocated according to the type of allocate call and the parameters passed. There are three main forms of allocate call:

- upc_alloc(), in which all memory is allocated on thread 0;
- upc_all_alloc(), in which memory in an array is allocated in a round-robin fashion according to the blocking factor; and,
- upc_global_alloc(), which is typically used to allocated multiple arrays with the same dimensions, one array per thread.

The design of UPC allocations enables UPC to make appropriate use of NUMA memory layouts. When a shared array variable is allocated using upc_all_alloc(), a chunk of that array then resides in local shared memory. Assuming that the allocation and subsequent work sharing are done correctly, the thread will then spend the majority of its time accessing shared memory that is local to that CPU core. Pathological NUMA memory access cases are avoided. These pathological cases are ones in which most of a thread's working set resides in the physical memory associated with a distant CPU, requiring additional traversals of NUMA pathways and increasing memory latency.

3.1.2. Work Sharing

Work sharing is done primarily through the use of the *upc_forall* construct. This statement is used in place of the for statement on loops for which work sharing is to occur. The upc_forall statement is of the form:

```
upc_forall(expression1; expression2; expression3; affinity)
```

The first three expressions are equivalent to those for a normal C for loop. The affinity component indicates to a particular thread of execution which subset of the total loop iterations it should execute. In the simple case, if affinity is an integer expression, a thread will execute all iterations in which (*affinity* modulo *number of threads*) equates to the current thread number[1]. The simple case of using the loop counter variable as the affinity expression usually results in best performance.

[1] El-Ghazawi, Tarek, et al. UPC: Distributed Shared Memory Programming. Wiley-Interscience. P. 51.

3.1.3. UPC Threads and Network Layer

The implementation of UPC threads is not restricted to actual intranode userspace threads, and the two are distinct concepts. In the case of Berkeley UPC, UPC applications run on top of a layer called the GASNet (Global Address Space Network), and the GASNet determines the actual thread implementation. Possible GASNet layers include SMP (single node with pthreads), PSHM (process shared memory), MPI (in which the UPC shared memory is actually implemented via MPI calls), and various network APIs such as Infiniband verbs.

The number of UPC *threads*[2] is fixed at program startup, and does not change during the code's execution. This attribute of UPC makes it similar to MPI in that each process or *thread* is alive from inception through exit.

3.2. OpenCL Overview

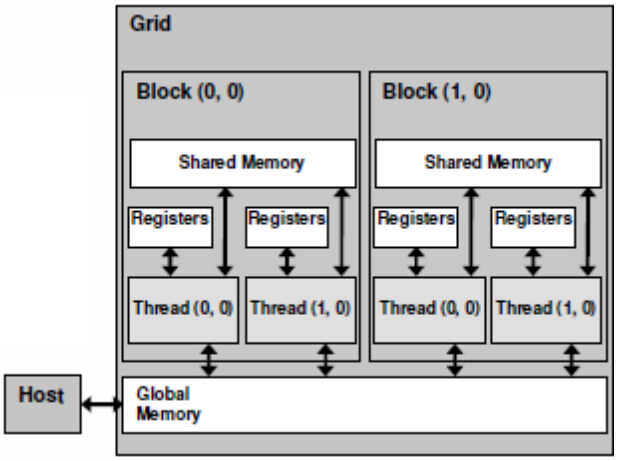
OpenCL (Open Computing Language) is a parallel programming ecosystem intended for use with heterogeneous processing environments. It is similar to the CUDA system, mentioned in the next section, in that it is able to target GPUs. However, OpenCL is more general-purpose than CUDA, and may be used on any device that has a supported implementation. Programs that utilize OpenCL consist of traditional code (C/C++), along with the OpenCL API, which enables the setup and control of execution *kernels*, which perform the computationally intensive work requiring parallelization. Kernels are written in a subset of the C99 language, and are compiled to target a computing device. Supported devices include (multicore) CPUs, GPUs, and accelerator devices such as the Cell BE. In this manner, multiple types of computing resources may execute binaries built from the same kernel source and using the same setup/communication code.

3.3. Cuda Overview

CUDA, *Compute Unified Device Architecture*, is a programming model and instruction set architecture initially released in November 2006 by NVIDIA to allow for application developers to access GPUs (Graphical Processing Units) without having to use the graphics API. CUDA comes with a software environment that supports C, along with Fortran, OpenCL, and DirectCompute. Additionally, you can get compilers for PyCUDA (Python) and JCUDA (Java-CUDA).

The core concepts for CUDA revolve around three key abstractions: a hierarchy of thread groups (think tree structure), shared memories, and barrier synchronization. These abstractions are accessible to the programmer through a set of language constructs. The programmer must think about data parallelism and the effects of threading when considering how to partition his problem because, as seen in Figure 3-1, data cannot be shared across blocks. Blocks are normally assigned to separate Streaming Multiprocessors (SMs).

[2] The term *thread* in UPC refers to an independent execution of the code. The underlying implementation of a UPC thread can be either a userspace thread or a process, depending on the underlying network layer.



[3]**Figure 3-1 Diagram of CUDA GPU Memory layout**

The overall methodology is heterogeneous computing with the CPU executing sequential portions while parallel operations are executed on the GPUs. When programming in CUDA, you must think about where entities will reside or need to be accessible from – i.e., will this data/function reside solely on the *host* (CPU) or will it go/be accessible to the *device* (GPU), which memory of the device will it use (global, shared, private), and so on. Then, you must allocate the data, copy the data to the GPU, compute on the GPU, then copy the data back to the CPU, using appropriate CUDA directives. The goal for optimized CUDA code is to copy the data as infrequently as possible and do as many computations on this data as possible once it is on the GPU.

[3]*VolumeI_CUDA_Intro.pdf*, http://developer.nvidia.com/object/cuda_training.html, vg 21

4. Application Selection and Suitability

4.1. Code Team Interviews

We interviewed a number of code teams and learned about the characteristics of their codes to determine if they would be suitable for our study. In addition to helping us better understand these codes and decide whether they should be included in the study, the interviews provided an opportunity for us to discuss exascale challenges in greater detail with some of the teams. We recommend that we hold follow-up meetings with these teams to discuss our findings and discuss further the languages and tools that should be utilized in their preparatory plans for exascale.

See Table 4-1 below for a description of these codes.

Code Name	Team Members	Selected	Code Name
LEOS	Burl Hall, Rob Neely, Tom Epperly, Dale Slone, Ellen Hill	Yes	The Livermore Equation of State (LEOS) Package was our initial code of interest. One of us (Evi) began working with LEOS prior to other team members joining the project. The Livermore Equation of State (LEOS) project generates and delivers equation of state data tables for use in LLNL hydrocode simulations. In 2006-2007, Tom Epperly explored the use of Global Arrays and a caching system[4] to reduce the memory footprint of this library. Since his work, the library has been rewritten. After discussion with team members, we decided it would be interesting to see how UPC, using the shared memory with affinity approach, would handle the large coefficient array prevalent in LEOS and a concern to multi-physics code teams as new computer architectures come to LLNL. We want to see if we can use UPC's shared concept to spread this coefficient array across threads/processors efficiently versus having the entire coefficient on every processor. To control the scope, we further decided to look at the Livermore interpolate Package, (LIP), and focus on the calculation of the coefficient array within this library since it is now a stand-alone package.
CP2K	Will Kuo	No	CP2K is an F95 code to perform molecular and atomistic simulations of substances in different phase states[5]. The code has components for traditonal molecular dynamics, density functionals, and Kim-Gordon models. Because no version of the code and no kernels are written in C, we deemed it an unsuitable candidate for this study. Will suggested that we meet instead with the Cheetah team, and that the kinetics and thermo routines within Cheetah would be of interest.
Cheetah	Larry Fried, Peter	No	Cheetah is a widely used thermo-chemical code written in C. We met with Larry, Soren Bastia, Will Kuo, and Peter Vitello to discuss

[4] *Scalable Equation of State Capability*, Tom Epperly, Fred Fritsch, Peter Norquist, and Lawrence Sanford, November 2007

[5] http://cp2k.berlios.de/

Code Name	Team Members	Selected	Code Name
	Vitello, Soren Bastia, Will Kuo		the possibility of using Cheetah as the basis of our study. Because the code is a collection of routines, and these routines must be called from outside Cheetah, we needed a 'standard' caller case for the purposes of our study. The team did have a wrapper case that they were able to provide to us. Due to the complexity of the code and other issues, we opted not to study Cheetah.
Hypre	Rob Falgout	No	Hypre is part of the scalable linear solvers project at LLNL. It is 'a library of high performance preconditioners that features parallel multigrid methods for both structured and unstructured grid problems.'[6]. It is a sparse matrix code written in C. In our meeting with Rob, he suggested that if we were to select Hypre, we focus on the AMG2006 (Algebraic Multigrid) solver. That solver is the benchmark for an LDRD project being worked on by Allison Baker, Martin Schulz, and Bronis deSupinski . It examines OpenMP performance when used in conjunction with a memory affinity patch written by that team.
PF3D	Bert Still	No	PF3D is a key code in use in the National Ignition Facility. PF3D is written in C and is used to model the behavior of the laser light inside the target chamber of NIF. A laser-plasma interaction effect makes PF3D simulations critical to NIF's success. As the wavelength and energy of laser light increase inside the chamber, and as the light is reflected off of the hohlraum , a natural process called Raman scattering inhibits measurement. This increases the need for accurate simulation with PF3D prior to conducting high-energy experiments. PF3D is highly scalable and there is a great demand for computational cycles to run simulations. The most important kernels in PF3D are contained in eight light wave solvers that look similar to each other. The vast majority of the code's time will be spent in these solvers for problems of interest. All data is stored in 1d heaps, and the code uses a regular Cartesian mesh. The PF3D kernels are relatively compact and the code builds fairly quickly. A complicating factor is that the code is run under Yorick, which is an interpreter also written in C. I (Charles) had initially begun working on PF3D, but switched to CLOMP in early June once we determined that CLOMP was better suited to an initial investigation. PF3D is a marquee application and contains critical loops that are well suited to work sharing. However, it is a much larger code than CLOMP. Parallelizing it

[6] https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html

Code Name	Team Members	Selected	Code Name
			with UPC would involve examining and likely modifying all functions in all code files. Yorick might also require modification. The PF3D code is currently parallelized with MPI and this adds complexity.
BLAST / blast+	Tom Slezak	No	In our meeting with Tom, we discussed the future needs of biocomputing applications like BLAST. Tom noted that the primary algorithm for biocomputing involves repeated hash table lookups – looking for a particular DNA sequence within the larger genome. This is a memory-intensive operation. A large amount of physical memory is needed to store the full genome that is used in the lookup. Many of the current LLNL supercomputers are not well suited to the sorts of scientific problems that Tom's team is solving because there is not enough memory available. The team has purchased a special SGI Altix machine that has half a terabyte of memory. This system, along with FPGA-based systems that are specially designed for solving the pattern-matching problem, are where computing is headed for bio. Pacific Biosciences will soon be charging very little to do sequencing. The reduction in cost is so significant that it changes the nature of the problem in bio, and analysis of results, rather than sequencing, becomes the major bottleneck. Tom is concerned that the future of generalized supercomputing, in which there is an abundance of CPU cores on nodes but no significant shift in memory per core, or even a reduction, is not the direction that biocomputing needs in order to solve next-generation problems. An example of a next generation bio problem is metagenomic sequencing, in which all of the DNA from a soil sample is analyzed to determine its origin and is compared against multiple genomes. The metagenomic problem will require 1 to 2 TB of RAM. Tom envisioned a metaphorical solution that had a biological appearance. In his solution there is a very large sphere of memory which contains the problem, and then compute engines 'walk' the surface of the problem sphere. While the global memory addressability of UPC might be theoretically useful for biocomputation, latency to memory is the ultimate limiting factor for applications like BLAST, and addressing off-node memory causes such a major performance reduction that the application is effectively unusable. Since only one task will be running per node in order to provide the maximum amount of addressable memory, there is little opportunity for task- or thread-level parallelism. Cross-node parallelism is

Code Name	Team Members	Selected	Code Name
			embarrassingly parallel.
CLOMP	John Gyllenhaal	Yes	C Livermore OpenMP benchmark. CLOMP is a benchmark code that mimics a key laboratory code. John Gyllenhaal and Greg Bronevetsky developed the CLOMP code to measure OpenMP overhead. The code is relatively small, is highly configurable via run-time parameters, and is ideal for modeling overheads associated with thread parallelism. In its unmodified form, the innermost loop in the code walks through members of a linked list. There are cross-iteration dependencies in the innermost loop, so thread parallelism is attained by work sharing of an outer loop that iterates over multiple linked lists which are independent of one another.

Table 4-1 Candidate code descriptions.

4.2. Selection Process

We decided which codes to study based on several factors:

- Either a key LLNL code or representative of key laboratory codes;
- Ease of build;
- Version already in existence with OpenMP directives to enable performance comparisons with OpenMP cases;
- Code has elements that we anticipated would 'push' or stress the language models tested;
- Code was compact enough to be ported to target language model in limited amount of time.

For these reasons, we selected the CLOMP code for our combined UPC, CUDA, and OpenCL porting effort.

LEOS was selected to represent a first physics porting effort based on some of these factors and also because it explored the use of UPC for the unified model of parallelism, as opposed to the hybrid model.

5. Learning the New Paradigms: Effort and Lessons

We hosted several activities to help us move forward in learning about exascale programming models:

- Lawrence Berkeley National Laboratory is where a large part of UPC is currently developed. We contacted Paul Hargrove, one of the lead developers, and arranged a visit. We met Filip Blagojevic , another developer who specialized in application codes. We learned more about UPC, and how UPC shared memory works. We heard how this effort is truly on a shoe string budget, and works with a minimalist approach. We discussed some of our concerns about needing to move shared variables to the top of the code, and heard that although these are real issues, there is little room for changes to the standard at this time. We learned there are local versions of UPC (i.e., the Berkeley version) that have features that get around some of the limitations to UPC. However, these features are not portable. We also learned that the MPI-UPC collaboration was done without help from LBL. Finally, Charles showed his coding issues, and Paul and Filip worked through some of the bugs. We then got his code to compile, learning in the process about UPC. Because of this connection, we now have a conduit into the UPC system that can get us answers to some of our questions.
- To learn more about what others at LLNL are doing concerning GPU programming and CUDA in particular, we hosted a meeting with a group of scientists and computer scientists doing CUDA work. These folks included Jon Cohen, John Gyllenhaal, Lee Nau, Charles Shererda, Evi Dube, David Richards, Jim Glosli, Tod Gamlin, Lukasz, and Manaschai Kunaseth. Jon Cohen gave a talk on the speedup he has made going from OpenGL to CUDA –which was quite impressive. It did seem that the projects were GPU specific. David Richards had two summer students (Lukasz and Manaschai) working with the same physics module, converting it to OpenMP and CUDA. Luskasz has good experience in CUDA, and he described his current project as his toughest to-date. He had similar issues similar to ours in porting, including handling pointers. Our plan is to stay connected via a majordomo list and the SharePoint site

We set up a SharePoint Site which allows us to maintain a place for documents and communication to team and outside groups. When Charles started working on the project, this site proved useful in having a handy place for him to locate all of Evi's detailed journals. Her research and journals helped him understand some of the pitfalls and challenges of UPC before he encountered them on his own, and reduced the time he spent looking for relevant information. The summer students have used this site to archive their information.

5.1. UPC

Both Evi and Charles came onto this project with a background in parallel and C programming but no prior experience with UPC. This gave us the opportunity to determine the relative difficulty of learning UPC, and what challenges a first-time UPC programmer faces.

We both began learning UPC by reading the book "UPC: Distributed Shared Memory Programming" by Tarek El-Ghazawi, William Carlson, Thomas Sterling and Katherine Yelick . It becomes apparent in learning UPC that there is not a vendor backing this language or a large number of people to support an infrastructure. Sources include the one book, written a few years ago, examples on the web from a few sources (all academic), a handful of PowerPoint presentations from the Supercomputing Conference from a number of years ago, and a handful of research papers, again all academic. In Appendix C, you can find more about these sources.

Concurrently, we each built and installed the Berkeley UPC compiler and translator and different GASNets so as to increase our understanding of the Berkeley implementation. Documentation for the build and install process, and for using the compiler and translator, is limited to READMEs and help text. A cautionary note: When building, it's necessary to verify you have successfully linked the compiler with the translator. Otherwise, your programs will be shipped unannounced to Berkeley to be translated - an undocumented feature. When determining the compiler variations, there are many types of GASNet options, and you must peruse the README files to decide which version you want to build. Additionally, you must figure out what the difference is between the versions and why you would want to chose one over the other. To-date, we cannot say why one is better than the other, and we have not run enough tests to understand the differences.

Once we had a working compiler and translator, we began to write and build simple UPC applications. It was only at this point that we really began to understand the UPC paradigm – that all UPC threads execute all program code and that the upc_forall construct splits the original loop into subsets of iterations that are executed by individual threads. This was also the point at which we gained familiarity with the runtime system and the memory management for UPC. Occasionally, there were errors in the book and PowerPoint slides, so patience and problem solving were in order, along with confidence in your ability to debug with printf statements, since debuggers supporting UPC do not presently exist. When actually writing programs, there are limited existing codes out there. We did use the websites from Michigan Tech University for examples.

Perhaps the most difficult aspect of UPC to learn is the UPC shared memory model. It is hard to figure out what is really happening shared memory-wise. Since there is no debugger, you have no way of knowing what is happening in memory apart from using printf.

5.2. OpenCL

There is a wealth of resources available for getting up to speed on OpenCL. Materials exist from the major vendors supporting the standard (NVIDIA, AMD, Apple), as well as from academic institutions. While not contained in one central location as with the CUDA resources, they are fairly easily located. Having previous experience with CUDA is definitely helpful, although not required for learning to use OpenCL. Learning from several sources was helpful in that resources were stronger in some areas than others, so they nicely complemented each other. In Appendix C is a list of resources used and their various strengths and weaknesses.

No one resource is the be-all and end-all of learning OpenCL. Some are tailored to more specific audiences and go into varying levels of detail. Also, different platforms may not behave identically. Code that compiled and generated no errors on the Apple implementation included with OS X 10.6 (Snow Leopard) did not function properly when run using the NVIDIA implementation. Thus, it is important to have thorough error checking and reporting code, since assumptions made for one platform may not hold on another. It is also helpful to run code on different platforms, to compare and contrast implementation differences, and to locate non-portable code.

5.3. Cuda

CUDA has been in existence longer than OpenCL, and is supported by a single vendor, NVIDIA, so the system has had time to mature. The learning resources available are thorough and robust. The CUDA developer web site is well maintained and contains useful examples and documentation. Some examples are partially coded already, and only requiring filling in specific parts to get started. These progress from easy to more challenging examples and so guide the learner as his skill increases.

There are some challenges in getting code to work in an HPC environment where jobs must be submitted to a queue, such as edgelet at LLNL. Most of the learning examples assume a user has a single workstation personally available to him, and that code will be run locally. Working in an HPC environment generally has two differences: loading the cuda module into the working environment, and submitting jobs to a queue to be executed.

Although CUDA is relatively stable, at least in comparison to OpenCL, it should be noted that there are new releases usually two to three times a year. In fact, the same week as authoring this, the newest release candidate 3.2 was announced. While this release schedule is not overly frequent, it is more frequent than some other more established software.

The concept of CUDA programming is similar to vector programming, with the addition of moving data to and from a GPU device before and after computation. In order to amortize these data transfer penalties, computation kernels which are intensive are desirable from a performance perspective.

The lack of pointer support for older hardware is an annoyance, and the CLOMP port was not able to utilize them, as the newest hardware did not arrive at LLNL until late in the summer. More generally, and perhaps more challenging than the frequency of software releases, is the rate of hardware releases, each of which brings performance and usability features, but also requires retuning and enhancement of existing codes.

6. Code conversion: Effort and Lessons

In porting any code, the first task is to have a firm grasp of what the code does and how it does it. A portion of the porting effort for our team was devoted to understanding the codes that we were porting. Presumably the time that we spent on this establishes an upper bound; we expect that the typical application developer will be more knowledgeable and familiar with the code that he is porting than we were initially with the codes we studied.

We believe, however, that there is a 'sweet spot' of code familiarity. If a developer is overly familiar with his code and has not changed it significantly in a long while, he runs the risk of becoming mentally invested in the existing code design. This can create barriers to forming creative solutions in code when new challenges such as exascale arise. The new language models will certainly require creative thinking on the part of application developers. More importantly, whatever the language model, the forthcoming hardware architectures and the massive increase in scale make creative approaches to problem solving an absolute requirement.

One recommendation we make independent of this study is for code teams to hold architecting brainstorms in which they consider how they would go about solving some of their key problems given scale and target architecture design, while ignoring the existing code base. This effort could also ignore programming language models and specific parallelism solutions, to the extent this is possible.

An outcome of a series of such brainstorms might answer the question:

> Given a particular scientific problem of interest, and an architecture of either GPU-based or many-core-based systems at a particular scale, what would a code look like that solved this problem and took maximal advantage of the architecture and scale?

Certain assumptions could be made to enhance the usefulness of the design, such as, the code must mix two modes of parallelism, one to take advantage of on-node parallelism and one such as MPI to take advantage of off-node parallelism.

Once this high-level design is complete, code teams could then compare it to the existing code base and see where overlaps and gaps existed. In some cases, large portions of code may need to be rearchitected to make full use of exascale. By holding these sessions early, however, the team can identify what the code architecture should be, jump-start the creativity process, and solve whole new domains of problems.

A further recommendation is that teams be reconfigured slightly to introduce fresh ideas and assist with the creative process. We recommend soliciting volunteers or 'exchange developers' to shift across code teams so that at least one new person and sometimes two new people change teams in order to participate in the design effort. These exchange developers should be staff who are ready to offer their perspective in spite of a lack of familiarity with the new code.

In this study, we focused our efforts on codes without MPI parallelism. All of our porting efforts either replaced existing parallelism or parallelized a non-parallel code. With only a single mode of parallelism in our codes, the study was strictly an examination of these novel forms of parallelism. We have not yet examined the effort involved in mixing these parallelism modes.

Mixing modes will introduce additional programming complexity. It will also create the potential for new forms of race condition and deadlock. There may also be special challenges associated with the network layer, especially if the network layer is shared across parallelism modes. We strongly recommend that the network layer not be shared. See Section 6.1.2 for further discussion.

6.1. CLOMP - UPC

6.1.1. Effort Required

The effort to convert CLOMP from OpenMP to UPC and get the code into a working state is detailed below in Table 6-1. Note that our initial meeting to discuss CLOMP was on June 4, and I (Charles) worked approximately half time at LLNL on this effort. Task durations are approximate and do not reflect the fact that the porting effort was a more integrated and iterative process than what this table would suggest.

Task	Approx. Duration	Notes
Familiarization with code (includes initial meetings with code author, sketching out a call tree and data usage chart to assist with comprehension, test runs, and experimentation to understand the properties of the code under different runtime scenarios.)	4 days	This figure will vary widely by programmer. I (Charles) was slow initially as I regained familiarity with HPC applications. For many this figure would be lower for the CLOMP code, and for some, higher. This code was relatively small. For a large parallel application, a very significant amount of time – months – could be spent learning the code, what the various components do, the build system, how to make effective use of it for different types of problems, and where the majority of time is spent for different types of problems.
Identification and removal of OpenMP pragmas and function calls[7]	4 hours	The actual time to do the removal in this code is trivial. More time was spent understanding what the various parallel code components were doing, such as identifying specific variables that were shared or private, and loops that were parallelized and would be parallelized under UPC.
Removal of extraneous OpenMP functions[8]	1 day	Some of this time was spent in

[7] To eliminate compiler warning messages, all OpenMP pragmas were removed. OpenMP function calls such as omp_get_num_threads() and omp_get_thread_num() were removed and replaced with the relevant references to UPC constants such as THREADS and MYTHREAD.

[8] The original CLOMP code contains OpenMP best case, static, dynamic, and manual parallel loops. It does not make sense to mix OpenMP and UPC in the same code, since both are threading paradigms, and in the case of UPC, all threads are alive for the duration of the code. Additionally, UPC inserts additional code for all shared memory variable references, so that code performance can be perturbed from the standard C case even if the

Task	Approx. Duration	Notes
		understanding what these routines were doing and whether there was any opportunity to preserve them or create similar routines using UPC.
Identification of variables to convert to shared memory variables	2 hours	Time spent identifying variables was minor. Note that if more time had been spent looking at this exhaustively, it might have saved later debugging time.
Conversion of variables to shared memory variables. Movement of any local variables that must be shared to global declaration section	2 days	Conversion of variables from standard variable types to UPC shared memory variables was conceptually challenging in several cases.
Proper allocation of shared memory variables	4 hours	While the amount of code to do this is trivial, it was necessary to carefully consider what form of UPC allocate calls should be used for each dynamic shared variable.
Conversion of for loops that must be parallelized to upc_forall loops	4 hours	Parallelization of for loops was not a simple one-to-one correspondence with prior OpenMP parallelization. Each loop was examined.
Serialization of all parts of code that must be performed in serial	3 days	It was not possible to simply serialize (conditionally execute if MYTHREAD = 0) all portions of the code that were run from a single-threaded region in the OpenMP case. This is because private variables must be initialized for all threads, not just for thread 0. This made serialization one of the more challenging tasks - updates to shared variables had to be either serialized or operated on with a upc_forall loop, but updates to private variables had to be executed independently by all threads.
Debugging of basic compilation problems	2 hours	Minor amount of time.
Debugging of runtime issues:		

code is run in a serialized region. For these reasons, comparisons between OpenMP and UPC runs of the CLOMP code must be with different executables rather than different timed test cases within the same binary.

Task	Approx. Duration	Notes
Serial code issues such as: segmentation violations; incorrect results obtained in serial case; failure to run to completion in serial case	3 days	There were a number of serial code issues, but the one that consumed the most time was incorrect results caused by a failure to advance a pointer.
Parallel code issues such as: segmentation violations; incorrect results obtained in parallel case; failure to run to completion in parallel case	13 days	Parallel run bugs and race conditions were caused primarily by the following issues: • Portions of code were serialized that should have been parallelized; • Portions of code were parallelized that should have been serialized; • Private variables should have been declared as shared, and vice versa; • Missing barriers where barriers were needed; • Wayward barriers in serial sections of code.
Performance analysis and tuning	3 days	Most effort was spent in analysis of C code generated by translator.
Total	**31 days**	

Table 6-1 Work breakdown, UPC learning and code conversion effort, CLOMP.

6.1.2. Overall Lessons

The lessons that I (Charles) learned from porting CLOMP to UPC are:

- UPC basics are easy to learn;
- Pointer arithmetic causes code insertion;
- It is challenging to fully conceptualize complex data layouts as shared types;
- Shared variables always have global scope;
- Porting code that is already threaded is nontrivial; and
- Mixing MPI and UPC will be very challenging.

These are explained further below, along with their implications.

UPC Basics Are Easy to Learn

It is fairly easy to acquire the basics of UPC. Unlike MPI, there is not a large library of function calls to learn. However, understanding the idiosyncrasies of UPC, especially allocation and manipulation of shared variables, takes practice and skill.

Pointer Arithmetic Causes Code Insertion

UPC lends itself well to computational problems that can be modeled using a single global address space. In the case of the CLOMP code, each thread operates on a group of linked lists that are independent of each other. This is not the most advantageous case for UPC because there is a fairly

high performance penalty for doing pointer arithmetic in UPC: the UPC translator inserts two function calls for every pointer dereference and modification. Those functions are inlined but each calls other functions that interact with the underlying GASNet layer. This results in a significant amount of total code inserted. See Appendix A for an example of the code that is inserted by the UPC translator.

It Is Challenging to Fully Conceptualize Complex Data Layouts as Shared Types

The layout of an important data component in the CLOMP code is implemented as a pointer to a pointer to a data structure. Mimicking this layout in UPC required the use of a shared pointer to a shared pointer to a shared data structure. This was a confusing layout and was the source of several bugs. I (Charles) spent a good deal of time considering the full implications of this layout. I mapped this out several times and had to put numerous printf's in the code to debug some problems with it. The first layout that I devised was also the final one that I used; however, in between this first and final stage I tried a number of alternate layouts while debugging. I was uncertain whether my original layout would work as I expected it to. The original layout was acceptable, and the bugs had different causes. In this case I spent time trying alternate approaches because of uncertainty about the language, rather than spending time looking for bugs in my code. This pattern was reinforced by some uncertainty on the part of the Berkeley UPC team about whether my data layout was acceptable. Better documentation and a more robust product would have enhanced my confidence that my coding approach was a legitimate one.

Shared Variables Always Have Global Scope

Shared variables are always global in UPC. They are declared at the beginning of a program. This is an impediment to porting a large code to UPC. It will require programmers to forgo modularity in their code design for any variables that reside in shared memory.

Porting Code That is Already Threaded is Nontrivial

Because all threads are alive from beginning of program, it is nontrivial to port code that is already threaded. Every piece of the code must be examined to determine if it should only be executed by thread 0, or if it should be executed in parallel. Because of the way shared variables are allocated and the need to populate private variables for each thread, there is not a one-to-one correspondence between serial versions of the code using a typical threading model and serial versions of the code in UPC. Wherever shared variables are used, there is the potential to overwrite those shared variables with multiple threads. Wherever private data is modified, there is the possibility that all threads must make those same modifications, depending on how that data is used.

Mixing MPI and UPC Will Be Very Challenging

Porting existing MPI applications to a mixed MPI/UPC model will be especially tricky. All UPC threads and all MPI tasks are simultaneously alive throughout the life of the program. If we assume a model in which UPC threads are used on-node and MPI across nodes, and an on-node form of GASNet such as PSHM (Process Shared Memory) or SMP (pthreads) is used by UPC, the two models will not be in contention for the same network layer. This eliminates some complexity.

However, all MPI communication must be performed only by UPC thread 0. UPC thread 0 must also have a consistent view of all threads' memory at the point when it attempts to access shared memory. The easiest way to implement this is through the use of UPC barrier calls. If UPC will be used in conjunction with MPI on production LLNL application codes, we recommend that LC develop a programming standard and tutorial for mixing the two models. Application programmers will learn how

to mix the two models in a consistent way, and LC staff will develop their expertise as they create the standard.

6.1.3. Debugging

Debugging the UPC version of the CLOMP code was a very slow process, primarily because the only debugging tool available for current versions of UPC is the printf statement.

One of us (Charles) spent several days investigating the potential of using TotalView with alternate versions of UPC, GASNet layer, and debugger. TotalView had been supported with earlier versions of Berkeley UPC and TotalView, but was not officially supported in more current releases of either. While it was possible to launch UPC codes under TotalView and insert breakpoints, the breakpoints were ignored, and it was not possible to halt program execution. In discussions with the Berkeley UPC team, we learned that they were not surprised to hear of the problem, were uncertain whether TotalView would work, and had not tried using TotalView themselves with current releases of either TotalView or UPC. We discussed the possibility of getting Berkeley UPC supported in a current TotalView release with our on-site TotalView expert, Matt Wolfe. It seemed likely that the amount of time that it would take to add UPC support would exceed the amount of time needed to debug CLOMP even if we categorized the issue as urgent. We agreed to make the issue of UPC support a medium-priority issue.

I (Charles) made the decision at this point to move forward with more primitive debugging methods as opposed to installing old versions of Totalview and UPC in an effort to get a working debugger. In hindsight it may have taken less time to install these old versions and get TotalView working than it did to debug using printf statements. Debugging the parallel version of the code took the most time of any single task by far.

A working debugger will be necessary in a production environment.

6.2. CLOMP - OpenCL

6.2.1. Effort Required

Familiarity with the CLOMP codebase had to be first attained before taking any other steps. Understanding the nuances and program design decisions was crucial for later steps involved with porting. Beginning with simple OpenCL examples was useful also, since a basic understanding was requisite for any code porting. After that, identifying the portions of CLOMP that had to change, and those that did not, was a key step, since it was desirable to modify the existing codebase only as much as absolutely necessary. The lack of pointer support in OpenCL required a restructuring of the main data types used (linked lists) into linear arrays suitable for memory copies to/from an OpenCL device. Also, since the CLOMP algorithm is essentially a two-step process, one emulating an MPI call, it was necessary to transfer data to the compute device and back at each iteration, which required some thought as to how best structure the transfer code.

We estimate that Lee, who worked on the OpenCL to CLOMP port, spent approximately 35 days of effort on learning OpenCL and on the port.

6.2.2. Overall Lessons

Starting with simple examples and working up to the CLOMP port was a straightforward process. However, some unexpected snags were hit along the way. It is crucial to completely understand an algorithm and its data dependencies before attempting to port it to OpenCL, which has a fairly restricted processing model. For instance, CLOMP depends on performing double-precision floating-point

operations, which are only supported by a subset of compute devices, and must be enabled using an extension to the API. The lack of pointers presented an interesting challenge, and required more effort than probably any other aspect of the port.

6.2.3. Debugging

Debugging is a very challenging aspect of writing OpenCL applications. A first key step is to provide as much error checking and reporting code as possible, to catch problems that might not otherwise be obvious. Beyond this, there is not really much debugging support. Even printf is only supported as an extension on some platforms and devices. Reading forums of people who had similar issues was helpful, but ultimately a very careful visual inspection of kernel code is the best debugging tool available currently.

6.3. CLOMP - Cuda

6.3.1. Effort Required

When we first applied CUDA to CLOMP, it did not have pointers or recursion, so the linked list in CLOMP had to be converted to an array. Additionally, the mindset for CUDA is different from that of the traditional MPI/OpenMP programming paradigm and it takes a while to get in the right frame of reference, i.e. traverse through the linked list and do work on each piece of data versus bundle up your data, ship it off to the GPU, do work, then ship it back. We did not ever quite get there with the CUDA conversion of CLOMP. As is always the case, it is easier to start with a fresh, clean piece of paper than to try to shoehorn an existing algorithm into a new programming paradigm.

We estimate that our summer student, Lance Harris, who worked on the CUDA to CLOMP port spent between 25 and 30 days of effort on learning CUDA and on the port.

6.3.2. Overall Lessons

This exercise had some positives and some negatives. We immersed ourselves in the language and learned by doing a real problem in which we had answers. We could compare results between different programming models and contrast different styles, and in talking to others doing CUDA, we ran into similar issues with the language which validated our positive experiences and challenges with converting our real problem.

6.3.3. Debugging

There are several options for debugging with CUDA. CUDA toolkit ships with CUDA.gdb debugger, and a profiler, along with the documentation to figure out how to run the debugger. Additionally, NVIDIA has set up their infrastructure to allow for others to easily build tools. Totalview has come out with a beta version for CUDA, and Alliana has a commercial debugger for CUDA which we will evaluate shortly.

6.4. LEOS - UPC

6.4.1. Effort Required

LEOS is under Subversion, on the Livermore Computing platforms. At ~70K lines of code and ~200 files, there is a learning curve to understanding the basics of LEOS. Fortunately reasonable documentation exists to get a quick overview of how the code works, and the data is organized in one large data structure. The part of LEOS that would be affected by UPC was isolated in one package of LEOS, called LIP, which is the Livermore Interpolation Package. LIP consists of ~50 files and its own test packages that are the standalone version of the interpolation package in the LEOS access library.

There were two challenges to the conversion. The first challenge was augmenting the data structure in which the coefficient array resided. This data structure consisted of all of the information for the interpolant. As the coefficient data was calculated, the structure would then allocate memory within the data structure.

LIP_interp		
LIP_Style	setup_type	Coefficient setup type
char	*xname	Name of the first independent variable (LEOS:rho).
integer	Nx	Number of x-grid values
real8	*x	Pointer to array of x-grid values
char	*yname	Name of the second independent variable (LEOS:T).
integer	Ny	Number of y-grid values
real8	*y	Pointer to array of y-grid values.
char	*fname	Name of the function being interpolated
real8	*fval	Pointer to array of f-values on (x,y)-grid
real8	*dfdx	Pointer to array of df/dx-values on this grid
real8	*dfdy	Pointer to array of df/dy-values on this grid
real8	*twists	Pointer to array of d(df/dx)/dy-values
real8	*coeff	Pointer to interpolation coefficient array
LIP_meth	int_type	Interpolation type for coeff

Figure 6-1 LEOS Data Structure.

After talking with the UPC experts from LBNL, and looking at the very few examples in literature, I (Evi) decided to leave the *coeff* array within the structure and make it a shared pointer, versus pulling *coeff* out of the structure, and having to recode a significant part of LEOS/LIP, which relied on *coeff* being within *the LIP_interp* struct. Making that decision brought about another challenge, one I would have regardless of the implementation method – *coeff* and thus *LIP_interp* needs to be declared at the global level. To get around that, I ensured that it was properly defined in a header file, and the header file was ubiquitous.

As was the case with the CLOMP conversion, much of my time was spent checking and rechecking myself concerning how to handle the shared memory, how was this data really going to be distributed across the threads, and did I correctly have the coefficient at the global level. I did not have confidence in myself, and had vague compiler errors that were not helpful. Luckily, we had the connection with LBNL at some point during the process so I could sent them snippets of code and they were able to help with the compilation of LIP.

Next, I had to convert a test problem to UPC, and I chose *liptest.c*, which tests the setup functions for LIP, building several cases of the coefficient array. My initial foray through *liptest.c* did not pick up on the fact that *liptest.c* used a temporary coefficient array to build the permanent coefficient array in the *Lip_interp* structure. I pulled the temporary coefficient up to global status and declared it to be a shared variable, finally getting the test problem to run.

Task	Approx. Duration	Notes
Familiarization with code - **LIP** (includes initial meetings with code author, and code team, sketching out a call tree and data usage chart to assist with comprehension, learning SourceForge and Subversion to gain access to LEOS/LIP)	20 days	Once I started asking questions about LEOS/LIP, realized that I was going to have to gain access to additional resources, learn to use SourceForge to get to documentation and the learn Subversion to get to the sourcecode. Additionally, since UPCC ships the code off-site, LIP would either have to be reviewed and released (time-wise infeasible) or I would have to figure out the error in compiling the translator on-site – another to-do. After LBNL visit, worked through LEOS coding again to set up the correct non-local shared information, i.e. have shared information at the highest level, and got that version of LEOS/LIP to compile.
Identification of variables to convert to shared memory variables	2 hours	
Conversion of variables to shared memory variables. Movement of any local variables that must be shared to global declaration section	8 hours	Finally figured out how to allocate memory across the threads by looking at several sources including the book and a few other sites. Drew out the memory for LEOS, and documented it.
Proper allocation of shared memory variables	10 hours	
Conversion of for loops that must be parallelized to upc_forall loops	8 hours	Non-trivial – look at all loops for *coeff* and figure out how to augment for shared memory
Debugging of basic compilation problems	5 days	Compiler messages are vague and take time to figure out.
Familiarization with code - **Liptest.c**	5 days	Took some time to figure out there was a temporary coefficient array that needed to be promoted and moved into shared memory, which required more thought and coding. The rest of the UPC transition went quickly since I was already familiar with the syntax.
Identification of variables to convert to	2 hours	

Task	Approx. Duration	Notes
shared memory variables		
Conversion of variables to shared memory variables. Movement of any local variables that must be shared to global declaration section	1 hour	
Proper allocation of shared memory variables	1 hour	
Conversion of for loops that must be parallelized to upc_forall loops	1 hour	
Debugging of basic compilation problems	1 hour	
Running several threaded problems	8 hours (and counting)	Have run and compared against two test problems in repository: First one thread versus serial, then several threads against serial.
Total	37 days	

Table 6-2 Work breakdown, UPC learning and code conversion effort, LEOS.

6.4.2. Overall Lessons

There is a learning curve with the application - LEOS/LIP, and there is a learning curve with UPC more than just adding C parallel directives. Although initially there appeared to be a reasonable number of web sites and a book to learn about UPC, until you actually try to apply UPC to a real problem versus running some of the test problems, there is not much help out there – you are on your own to explore and learn. Visiting LBNL helped a bit with making progress. The UPC Clomp section on Overall Lessons does an excellent job of summarizing many of the issues as does the breakdown above concerning where time was spent on the effort to port LEOS/LIP. Fortunately for this problem, the coefficient array, although a part of a structure, was global to the source code via an existing header file so I did not have to permeate the code with its presence. In the test case, for the temporary coefficient array, I had to think how to cleanly pull out the temporary array, then assign a local array per thread so as not to touch numerous functions with passing this shared array – non-trivial to do in UPC. It took me a while to think through how to minimize damage to the existing infrastructure with the UPC caveats.

Now I am trying to run with multiple threads, which need to be compiled in thus changing the Makefile and recompiling all of the source codes. I am having trouble with the compile, and have a vague error, so I need to try either google to find a similar error (no match), or send the error into the ticket system or LBNL contacts and wait for help.

6.4.3. Debugging

Debugging consists of printf and lots of swearing. Sometimes, you send snippets of code to LBNL, and they are able to help. Occasionally, you google, and you actually get a hit – about 5% of the time.

7. Results

7.1. CLOMP – UPC

The charts in this section are comparisons of OpenMP and UPC performance of the CLOMP code on the hera system for different input problems. Each node of hera has 4 Quad-core Opteron 8356 2.3 GHz CPUs for a total of 16 cores per node, and 32 GB of memory. All runs were performed on a dedicated, 'quiet' node.

OpenMP cases were compiled with the Intel 10.0 compilers with–O3 optimization. UPC cases were translated to C using the Berkeley 2.10.2 compiler/translator, and the Intel 10.0 compilers were then used to compile the translated code. All UPC cases used the PSHM (Process Shared Memory) GASNet layer for parallelization.

Speedup figures provided are relative to the OpenMP serial reference case for the given input problem. In Figures 7-1 through 7-3, all OpenMP threads allocate their own memory. This 'intelligent allocation' strategy optimizes NUMA memory access patterns and mimics the performance of the OpenMP memory affinity patch being developed by B. deSupinski, M. Schulz, and A. Baker at LLNL.

Over a variety of problem sizes, the raw time to run parallel regions of the UPC port of CLOMP was faster than dynamic OpenMP scheduled loops but slower than manual or statically scheduled OpenMP loops when shared memory for the OpenMP cases was allocated by each worker thread. Combined with the generally slower serial regions of the code under UPC, the present UPC port of CLOMP is significantly slower than an OpenMP port with an intelligent allocation strategy.

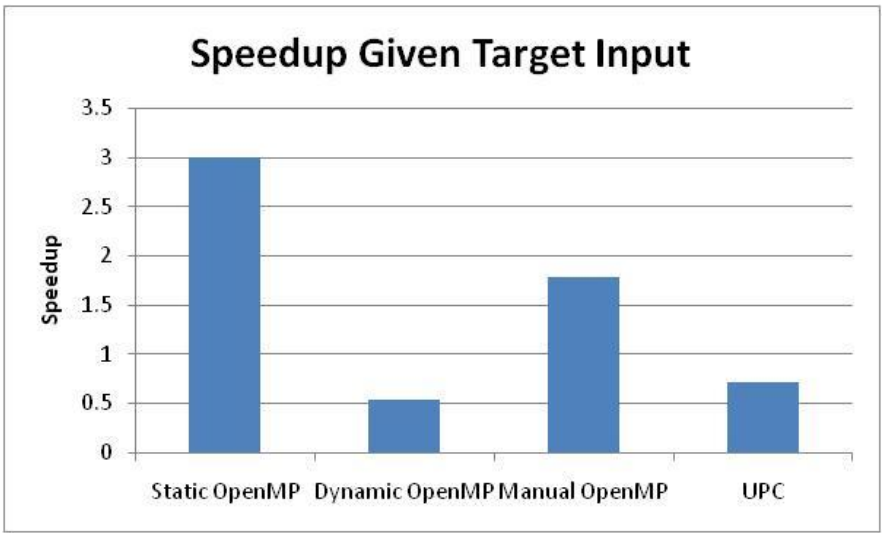


Figure 7-1 Speedup on 16-way hera node relative to OpenMP serial reference case for 'Target' input.

The 'Target' input case that was examined in Figure 7-1 is a small memory footprint problem (209k). There are 64 partitions and 100 zones per partition. This translates to 64 independent linked lists with 100 elements in each list.

Performance of all cases is low relative to peak speedup of 16 and the 'Bestcase' OpenMP speedup of 11.5. Bestcase speedup provides an upper bound on speedup but does not have adequate barriers to ensure correct answers. For this case and the cache-friendly input case (Figure 7-2), several OpenMP

environment variables were set in order to increase performance of the Static and Dynamic OpenMP cases, at the expense of the Manual case (and the Bestcase)[9].

Note that UPC performance is significantly worse than the serial reference case, as seen by a speedup value less than 1.

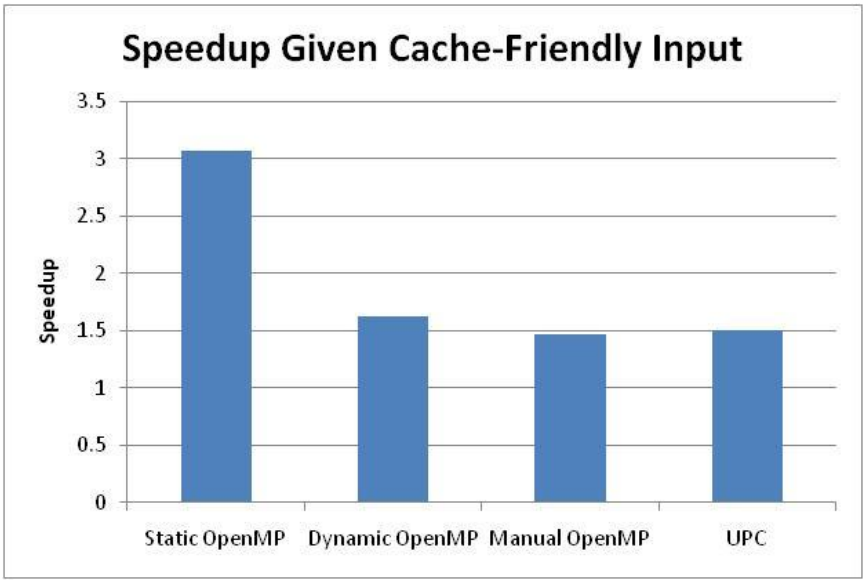


Figure 7-2 Speedup on 16-way hera node relative to serial reference case for cache-friendly input.

Figure 7-2 shows speedup for a very small memory footprint problem (6,656 bytes). Speedup for all cases is again poor relative to the OpenMP Bestcase of 12.5. For this problem, UPC does gain over the serial reference case, but still performs worse than the Static OpenMP case.

[9] These environment variables settings are: KMP_BLOCKTIME=infinite, KMP_LIBRARY=turnaround, and KMP_AFFINITY=compact,0

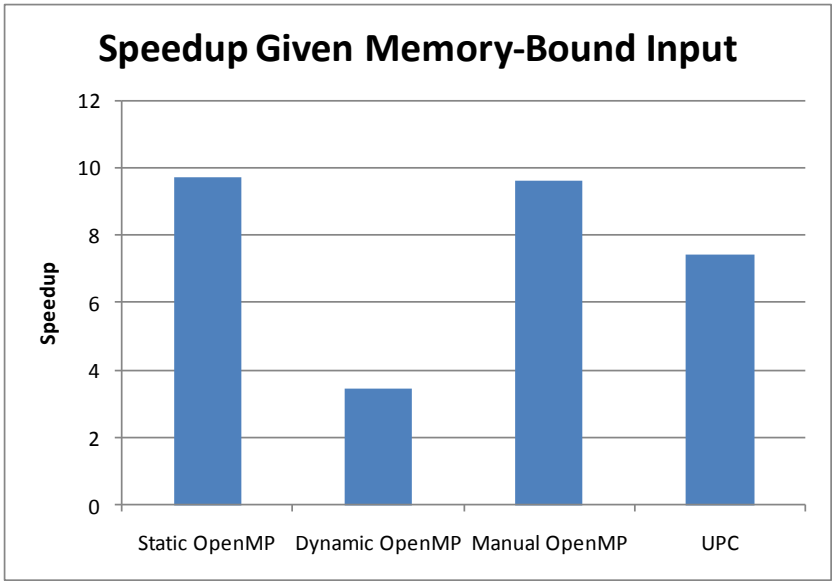


Figure 7-3 Speedup on 16-way hera node relative to serial reference case for memory-bound input.

Best results under UPC are seen with large-memory cases. This can be attributed to the lower relative impact of the code that is inserted to do pointer arithmetic on shared variables, which has a greater relative impact on small-memory cases or cases with a high flop/memory access ratio.

Figure 7-3 shows speedup for an input problem that is 328 MB in size. The larger memory footprint is achieved by increasing the length of each linked list to 10,000 zones. Speedup of all cases is closer to the Bestcase speedup of 14.8 than for smaller memory inputs. Despite performing better, the UPC case still lags in performance relative to the Static and Manual OpenMP cases.

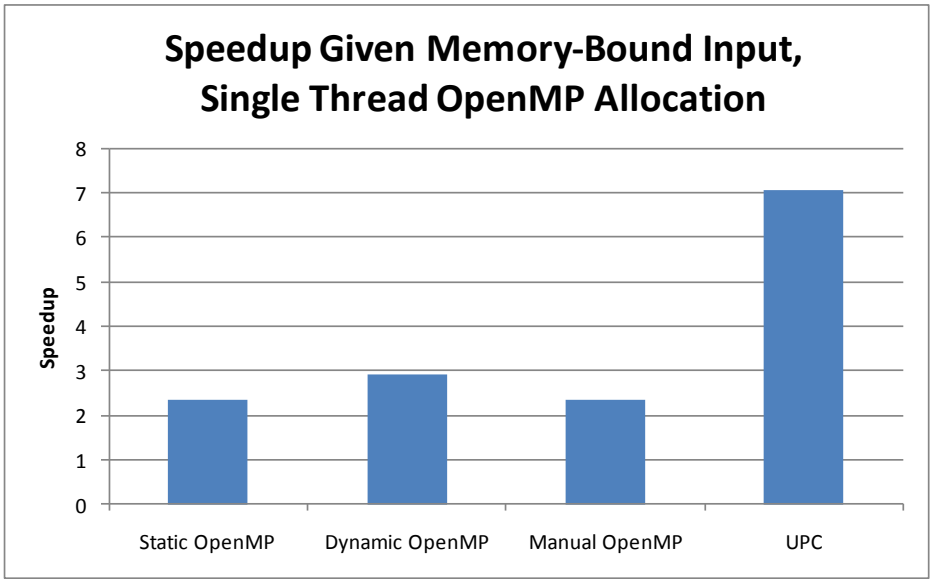


Figure 7-4 Speedup on 16-way hera node relative to serial reference case for memory-bound input and single thread allocation for OpenMP cases.

Figure 7-4 shows results for the same problem dimensions as Figure 7-3, but with a different allocation strategy for the linked lists in the OpenMP cases. In this case, OpenMP memory allocation is not done in a parallel region, so only one thread is used to allocate linked list memory. This allocation strategy

creates suboptimal memory access patterns on a NUMA system. The UPC allocation strategy for the linked lists remains the same as in Figure 7-3 and is a call to upc_all_alloc().

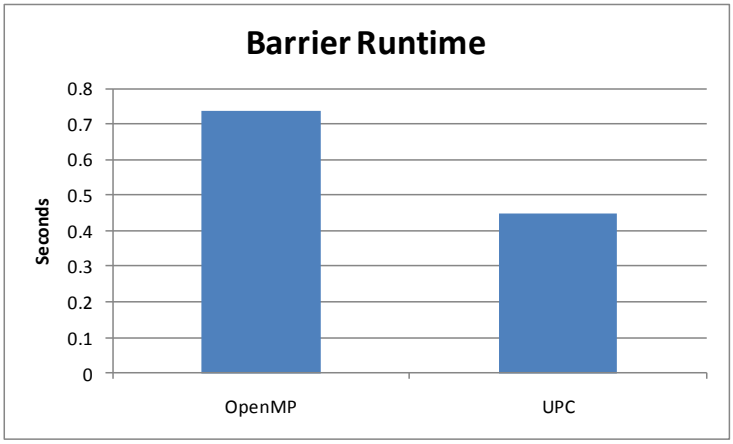


Figure 7-5 Runtime for Barrier loop.

As seen in Figure 7-5, UPC barriers are notably faster than OpenMP barriers for the compilers and platform tested.

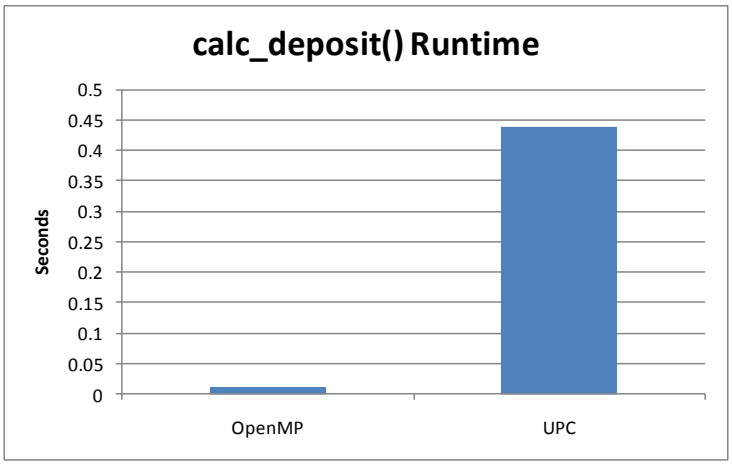


Figure 7-6 Runtime for calc_deposit() routine calls.

The calc_deposit() function emulates an MPI data exchange but does not do any actual communication. There are a large number of shared memory accesses that occur in this routine. Each shared memory access causes the UPC compiler to insert code. As a result this function takes 35 to 36 times more time in UPC code than the same routine coded in straight C/OpenMP, regardless of the problem dimensions.

There is a small amount of actual work in the original calc_deposit routine, and there are many function calls inserted. The actual number of top-level functions that are inlined into the code is equivalent to:

13 + (3 * numParts)

where numParts is the number of partitions, which is equivalent to the number of independent linked lists. This accounts for the high number of code insertions and the poor performance under UPC.

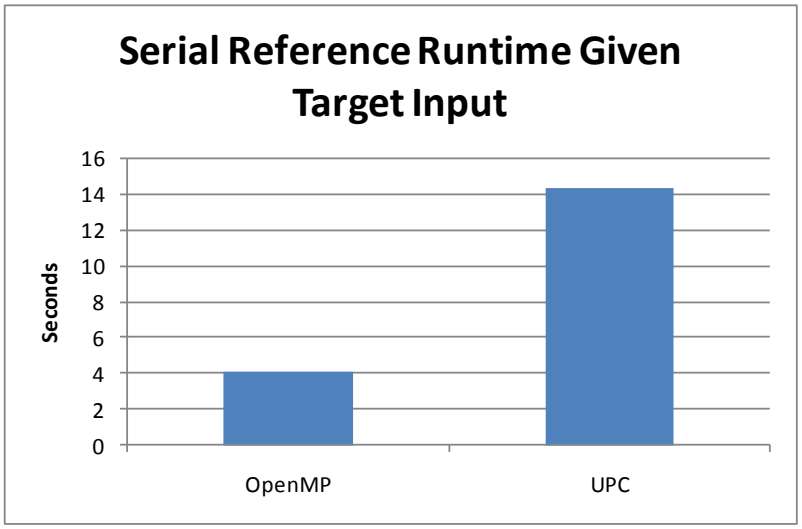


Figure 7-7 Runtime for serial reference case, OpenMP versus UPC.

Serial code performance under UPC was initially very poor due to use of the strict memory model. This improved with use of the relaxed memory model, but was still significantly slower than the OpenMP case regardless of problem size. The strict memory model inserts a number of additional barriers and function calls into the translated code. See Appendix A for an example of inserted code.

In Figure 7-7, the code insertion overhead shown for calc_deposit() in Figure 7-6 is apparent in the serial reference case. For the 'Target' input problem, the UPC case takes 3.5 times longer to run than the serial case written in C (with OpenMP in other regions of the code).

7.2. CLOMP - OpenCL

The OpenCL port of the CLOMP benchmark did not result in very impressive performance. In most cases, the OpenCL version was in fact slower than the serial version with no parallelization whatsoever. This was expected, however, since a great deal of buffer copies to/from the compute device were required. Even a version which minimized these copies was not much better than the serial version, and in many cases worse. It should be noted that no optimization steps were taken in the kernel code, which might have improved the speedup. The parameters lending themselves to the best speedup increased the problem size, and especially the number of floating-point operations to be performed. This mitigated some of the inefficiencies associated with OpenCL overheads such as buffer copies and kernel invocations.

Below are four different experiments measuring speedup relative to the serial (sequential) case. Each experiment has a different input type used to characterize different performance metrics. That is, different parameters were used with CLOMP in order to stress different parts of the system. These were run on edgelet, a system with two six-core Intel Xeon 2.66 GHz Westmere CPUs, 48 GB RAM, and two NVIDIA Tesla M2050 GPU Compute Units per node. The OpenCL cases were run on the GPU units, while the OpenMP (OMP) cases were run using the CPUs.

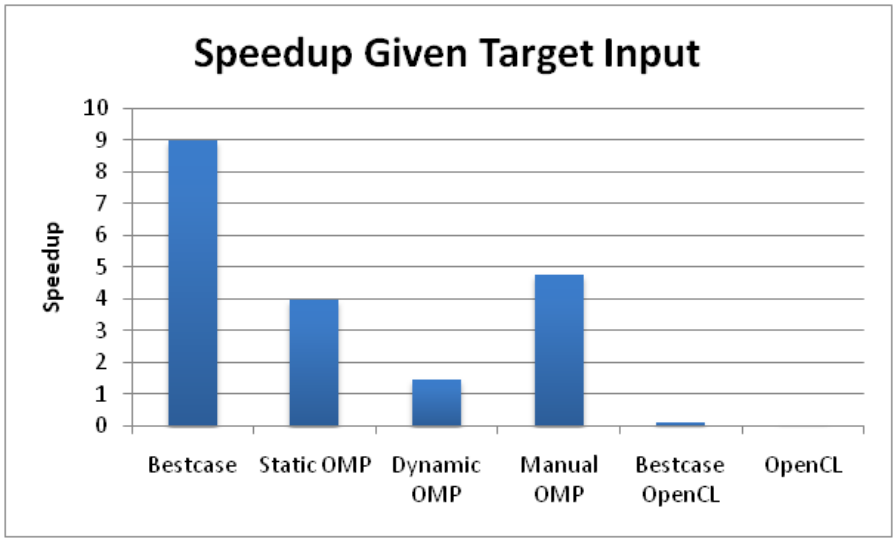


Figure 7-8 Speedup on 12-way edgelet node relative to serial reference case for target input.

Figure 7-8 provides performance analysis for the "default" case, which uses the target input parameters. Performance was very poor for the OpenCL cases, due to the many iterations being performed and high overhead.

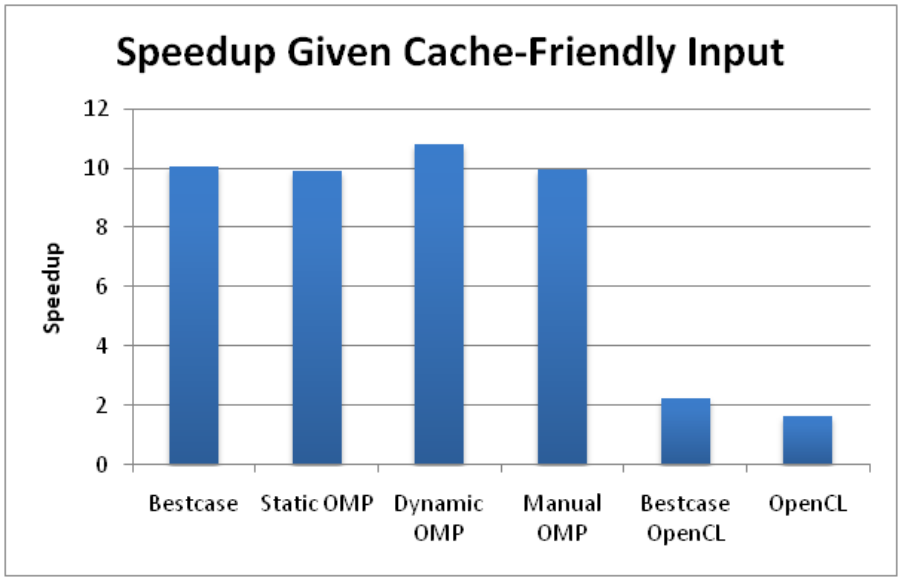


Figure 7-9 Speedup on 12-way edgelet node relative to serial reference case for cache-friendly input.

The cache-friendly version shown above in Figure 7-9 results in better performance for all cases. The CPU-based OpenMP cases were given more data (floating point operations) to calculate, which resulted in more cache resident data. The GPU-based OpenCL cases resulted in a two-times speedup over the serial case, also due to more floating point operations to be performed per iteration. We suspect that the dynamic OpenMP case performs better than the other OpenMP cases for this input because of load imbalances created by the thread count of 12, which is not a power of 2. The load across CPUs is therefore not evenly distributed, and the dynamic OpenMP case alleviates this issue with load balancing.

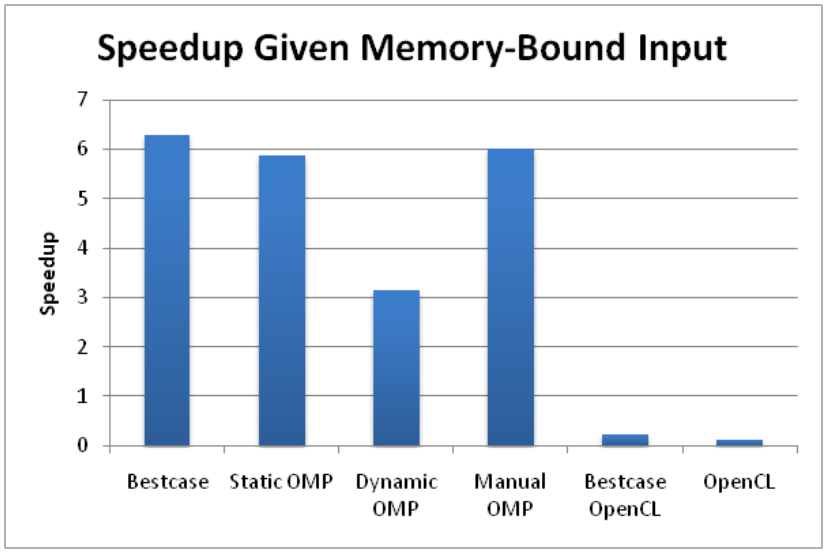


Figure 7-10 Speedup on 12-way edgelet node relative to serial reference case for memory-bound input.

The memory-bound input shown in Figure 7-10 consists of larger input data sizes to be operated on (more zones per partition). For the CPU tests, performance was worse than in the cache-friendly case, but worse than the target case. The additional data allowed more processing to be done in parallel, but still did not match the more flops-intensive case. The GPU tests performed very poorly, similar to the target case.

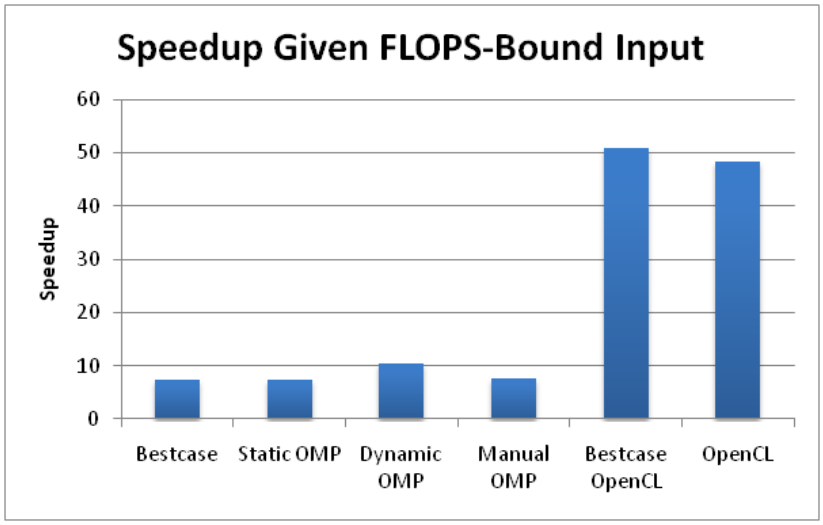


Figure 7-11 Speedup on 12-way edgelet node relative to serial reference case for flops-bound input.

The flops-intensive case shown in Figure 7-11 increases the amount of floating point operations per iteration by a factor of 10 over the cache-friendly case, also resulting in an overall decrease in iterations (since the time scale is not altered). The CPU cases are all comparable, and finally the GPU cases show significant speedup. This is due to the overhead of OpenCL and GPU memory buffers being mitigated by the sheer amount of computation to be done in each iteration. The raw floating point capability of the GPU in this case is clearly demonstrated.

7.3. CLOMP - Cuda

At this time, we are still in the process of converting code, and will continue to try to get the code compiled and debugged. As this was a learning process, we accomplished that task, and learned about CUDA and programming for a GPU. We have developed several guides, including a guide on tutorials available and a journal on how to compile/run CUDA.

7.4. LEOS - UPC

UPC cases were translated to C using the Berkeley 2.10.2 compiler/translator, and the gcc 4.3.2 compilers were then used to compile the translated code. All UPC cases used the SMP (pthreads) GASNet layer for parallelization.

Currently, timing is the only concrete result I(Evi) can report. As the problem size increases, I would like to find a way to report memory size per thread to verify distribution of the coefficient array.

Compiler	Liptest 1	Liptest4
gcc	elapsed time 0.080000	elapsed time 0.060000
upcc	elapsed time 0.080000	elapsed time 0.040000

I am surprised to see that the upcc compiler is faster than the gcc compiler for the Liptest4 case.

8. Code Maintenance and Development Environment Requirements

8.1. UPC

8.1.1. Application Team Requirements

In this section we describe what is needed in order to maintain the ported code, from an application developer's perspective, that was not required prior to the port, and separate from the items mentioned in Sections 8.1.2 and 8.1.3.

The primary requirement for code that has been ported to UPC is that application developers learn UPC and maintain familiarity with it. In order to make best use of UPC, a programmer must understand the problem data well enough to make decisions about how to divide it amongst all the threads in a way that maximizes data locality. As mentioned previously, we found aspects of the memory management concepts of UPC difficult to learn. This will likely be a challenge for application development teams as well.

Mixing UPC and MPI is a paradigm shift in parallel programming. Two separate modes of parallelism are both alive from program launch to program exit. Learning how to properly utilize both UPC and MPI without introducing conflicts and bugs will present another learning challenge to application developers.

Developers will likely require LLNL-specific UPC documentation along with support from an on-site UPC expert as they acquire the skills to program mixed MPI/UPC code.

8.1.2. GASNet (Global Address Space Network)

As discussed in Section 3.1.3, the GASNet is the layer that allows UPC compiled code to manipulate shared variables. It provides an API to compiled code and then communicates over the target network. This network could be pthreads, PSHM (process shared memory), or any of a number of network protocols such as Infiniband VAPI.

Multiple GASNets may have to be installed and supported in a production environment in order to provide flexibility. From a support perspective, it would be best to have as few GASNets installed as possible. In addition to supporting multiple installs, staff may have to devote time to isolating bugs in the GASNet layers as they arise.

The most likely scenario for mixing UPC and MPI in LLNL codes is to have UPC provide on-node parallelism and MPI provide off-node parallelism. If this model is followed, only on-node GASNet layers need to be installed. This would restrict the GASNets to PSHM and SMP (pthreads) and eliminates the problem of possible network contention between MPI and UPC.

8.1.3. Development Environment

8.1.3.1. Compiler and Translator

Building and maintaining UPC consists of installing a compiler, translator, and include files that must be accessible to the user. Additionally, there are a number of options available to the user regarding which underlying networking protocol can/should be built to optimally use the platform. The system then needs to be verified, ensuring that it has properly linked the compiler with the translator, and that the translator that exists at the LBNL web site will not be inadvertently used. A maintenance headache can arise if multiple versions of the compiler are built and maintained. Our advice is to settle on one or two versions per platform to minimize maintenance and developer choices.

The more pressing need is for expertise in UPC. As was mentioned in the above use case studies, it is non-trivial to gain expertise in UPC. The support infrastructure is limited which means enhancing the one we have developed and practicing with more use cases. Unlike the GPU community, there is no UPC user group to bounce questions off of or to get ideas from to assist with design or debugging, so you will be building your community on your own.

8.1.3.2. Debuggers

At present there is no debugger support for a current version of UPC. This will have to change if UPC is to be used at scale for LLNL applications. We have a medium-priority request filed with the TotalView team to support a current UPC. Ongoing support will involve testing new releases of TotalView and UPC to ensure nothing breaks, along with supporting bugs that users encounter while debugging UPC applications.

8.1.3.3. Performance Tools

There are several performance tools available for UPC from the University of Florida, such as the Parallel Performance Wizard (PPW). We did not test these tools during the course of this study. Basic support would include building and maintaining these tools and providing basic documentation on how to access and use them. Additional support would include responding to bug requests related to the tools, and providing expertise in the tools at user request.

To understand performance issues and bottlenecks in the CLOMP code, I (Charles) examined the translated code and compared it to the original serial C code. No additional tools are required beyond the default compiler and translator to generate and examine translated code.

A major cause of slow UPC code performance is excessive function calls and extra code inserted by the translator. Each reference to a shared memory pointer variable inserts a nontrivial amount of code. See Sections 6.1.2 and 7.1 as well as Appendix A for further discussion.

8.2. OpenCL

8.2.1. Application Team Requirements

Developers wishing to learn and/or use OpenCL have a variety of resources available to them. First and foremost, the Khronos group maintains the OpenCL specification and has materials available on their web site: http://www.khronos.org/opencl/. Additionally, there are presentations available from recent conferences, and from NVIDIA and AMD, on their respective web sites. There is also a book on OpenCL, entitled *The OpenCL Programming Book*. Information is available from the web site: http://www.fixstars.com/en/company/books/opencl/.

No official membership to any group or forum is required. However, depending on the hardware target platform(s), it would be wise to follow the forums and web sites of the vendor supplying the implementation. This will allow any device-specific idiosyncrasies or bugs to be more easily located and explained. Otherwise, the open nature of the platform allows for a variety of perspectives and resources, and no single source of information may be the only one needed. It is recommended that teams communicate with each other and other users tackling similar problems using OpenCL.

8.2.2. Development Environment and Compliers

OpenCL is a *specification* rather than a specific compiler/tool/platform. Therefore, any device or hardware platform may support the specification, but this requires an implementation that conforms to it.

Therefore, the development environment must be supported by an OpenCL implementation. The availability of this depends on the hardware and operating system used. That is, given a set of hardware, it must be determined whether the OpenCL prerequisite software has been released for such a platform. This depends on the hardware vendor, as the implementation must be customized for their hardware. This is different from the GNU toolchain for instance, which is supported across a variety of hardware platforms.

Currently, Apple, AMD, IBM, and NVIDIA have OpenCL implementations, with Intel promising a release "soon." So, while support depends on vendor implementation, most of the major ones have already released working versions.

In the case of the OpenCL version of CLOMP discussed in this paper, the Apple and NVIDIA implementations were used. The Apple implementation was used on a workstation to initially develop the code (and basic examples). No support at all is required, since an implementation ships natively with the newest Mac OS X 10.6. However, for the "production" version, the NVIDIA implementation was used, since NVIDIA GPUs were the hardware target. The software was pre-installed on the edgelet cluster, and includes the CUDA Toolkit and Developer Drivers.

8.2.3. Debuggers

The gDEBugger CL tool is a new one, which enables the debugging of OpenCL kernels. It is currently free and multi-platform, but is not really meant for an HPC environment. Since the GPUs used for this paper were accessed remotely, this tool is only minimally (if at all) useful.

The ATI Stream SDK supports debugging OpenCL kernels, but only in the x86 CPU case, and only using their platform implementation. Again, this is not useful for the environment used in this paper, since NVIDIA GPUs were used.

NVIDIA Parallel Nsight is a Visual Studio plugin that allows the debugging and profiling of OpenCL kernels, as well as CUDA (see next section). However, this is a Windows-only solution and therefore not useful in this case.

8.2.4. Performance Tools

The ATI Stream SDK includes tools to perform profiling and performance analysis of OpenCL programs. NVIDIA Parallel Nsight (mentioned above) also supports profiling. However, both of these tools are Windows-only and therefore not useful in this case. The NVIDIA Visual Profiler holds some promise, since it is cross-platform. However, it is still only suitable for a desktop setup, and not intended for cluster users.

8.3. Cuda

8.3.1. Application Team Requirements

As was mentioned in an earlier section, the NVIDIA Developer's Zone, found at *http://developer.nvidia.com/object/gpucomputing.html*, keeps tabs on the latest developments regarding CUDA and CUDA downloads and documentation. Forums, workshops, the latest news, blogs,

etc. are available, and you can register for a *GPU Computing registered developer account* which will give you up-to-the-minute information on NVIDIA's releases. This social networking capability allows an application developer access to and continuous feed of the latest information in a streamlined fashion, and the developer can filter information, passing toolkit updates to LC as needed. I thought it was an impressive way to use current social media at whatever level the developer wanted to engage at – and it showed me that this product is heavily endorsed by the vendor and is becoming production ready.

8.3.2. Development Environment and Compliers

We are impressed with the on-line tools and the web site and social media available – it certainly made it easier for several of us to learn about the programming model and come up-to-speed relatively quickly. I (Evi) had some minor issues with finding items, and the language has some idiosyncrasies, however all of them will be different, and that is part of the learning curve.

8.3.3. Debuggers

CUDA toolkit ships with CUDA.gdb debugger, and a profiler, along with the documentation to figure out how to run the debugger. AllineaDDT has a commercial CUDA debugger and Totalview is currently releasing their Beta version of their CUDA debugger. With the NVIDIA product, as long as the toolkit is loaded, we will have access to the new debugger, we just have to verify it works. With Allinea and Totalview, we need to maintain contact and collaborations with these companies. Currently, we have active alliances with these two companies to develop and improve existing debugger products, so adding CUDA to the mix should be a smooth operation since CUDA is used by many outside the National Laboratories (unlike UPC which seems to have a small following).

8.3.4. Performance Tools

CUDA Toolkit ships with a Visual Profiler that appears to get reasonable results from talks heard at the GPU Technology Conference 2010. This profiler gives a good first guess at your program's issues. Many scientists did suggest that stronger tools would be needed in the future to identify the more systemic issues with a program.

9. Conclusion

A lack of tools such as debuggers made accurate effort estimation for the UPC port difficult. We believe that the time we spent porting CLOMP to UPC was greater than the time an application developer would have spent on the same code, especially if he or she was already familiar with it. However, the present nature of the UPC language requires that an entire application be modified if one small region is to be parallelized. This implies that the time to parallelize a large code with UPC will always be longer than the time to parallelize with OpenMP.

We recommend against the use of UPC as an alternative to OpenMP for intranode parallelism in LLNL scientific applications that use MPI for internode parallelism. If an application has constructs that make the use of global arrays preferable to the existing code design, application teams may still choose to port to UPC. However, if an application design would not benefit from a switch to global arrays, we believe that the complexity involved in the port and subsequent code maintenance does not justify the performance increase relative to OpenMP. For a range of problem dimensions, parallel UPC code performance with the CLOMP code was worse than statically scheduled OpenMP parallelism, so long as an intelligent memory allocation strategy was used with OpenMP[10]. We do not know what the outcome of additional tuning might be, and we cannot rule out the possibility that such tuning could ultimately give UPC code an advantage over OpenMP for CLOMP. However, the large amount of code inserted by the Berkeley UPC translator for every UPC shared variable manipulation makes it seem unlikely that performance could be significantly better than OpenMP performance with memory affinity.

Table 9-1 compares UPC to OpenMP across a number of factors.

	OpenMP	UPC
Type	API supported by most modern C/C++ and Fortran compilers.	Language with compiler, translator, and GASNet layer that overlays C programming language.
Parallel programming model	Work sharing, mostly through parallelizing loops. Shared variables are updated by different threads in independent loop iterations.	Work sharing through parallelized for loops or thread-dependent execution paths. Shared variables updated through independent loop iterations.
Devices supported	Runs over multiple cores on a single node.	Flexible depending on GASNet layer. Can run over multiple cores on a single node, or across nodes by utilizing the system interconnect.
Threading model	Multithreading fork-join.	All threads independently run entire program.
Shared memory model	Shared variables can be accessed by any thread (restricted to one	Shared variables can be accessed from any thread of a run, which can extend

[10] A caveat: Determining behavior of applications at scale must at present be extrapolated from our intranode results. Only intranode parallelism was examined in this study, and no MPI runs were conducted.

	OpenMP	UPC
	node). Variable declarations and allocations do not change.	across nodes. Shared variables have unique forms of declaration and allocation and must be global variables.
Memory affinity	No default concept of memory affinity. Possible with affinity patch or by intelligent allocation.	Memory affinity available through use of particular allocate statements.
Performance	Without memory affinity: Fair With memory affinity: Good Using static loop scheduling: Good Using dynamic loop scheduling: Fair	Debug version: Very Poor Strict memory consistency: Poor Relaxed memory consistency: Good Calculations with excessive shared pointer arithmetic: Poor to Fair Memory-bound calculations: Good
Time to learn	Low	Low
Time to master	Medium	High
Time to port code	Low	High
Time to debug	Medium	High (will likely go down once debugger is available)
Support and documentation	Very good	Fair
Robustness and stability	Good	Fair

Table 9-1 UPC versus OpenMP.

We did not get far enough with our CUDA port to directly compare results to our OpenCL results. However, recent developments in CUDA as evidenced at the GPU Conference in San Jose during the week of September 20 show that it is becoming more robust, is supported by multiple vendors, is being experimented with by the scientific community with good results, and will soon be able to be used with the x86-64 architecture.

OpenCL proved to be a relatively straightforward model to use, once the verbosity of initialization and setup code was understood. The lack of UNIX-based cluster debugging and profiling tools was troublesome, but not completely inhibiting, thanks to the use of extensive error checks during kernel setup. Thanks to early multi-vendor adoption, there is a variety of information available about the standard, from beginning to advanced tutorials.

However, since this is still a relatively new standard, the lack of implementations (Intel is notably absent) was severely limiting for cross GPU-CPU comparison, which OpenCL could facilitate. All other major chip vendors have released an implementation. Such a release from Intel is promised to be soon, and would allow CPU vs. GPU performance analysis, using the same code. Additionally, it would facilitate an

OpenMP versus OpenCL performance analysis, both on the CPU, which would allow a true comparison of the two shared memory programming models.

It is expected that the standard will continue to mature and be adopted. Implementations will also become more bug-free and performance-minded as more users test and use them. Therefore, this programming model holds promise for future systems with heterogeneous parallel architectures.

	CUDA	OpenCL
Double-precision support	Yes, but depends on device	Yes, but depends on device and requires extension
Devices supported	NVIDIA GPUs only	Multiple, with implementation
Operating systems supported	Windows, Linux, Mac OS	Windows, Linux, Mac OS*
Languages supported	C, C++, Fortran, DirectCompute	C, C++
Pointer support	Yes, Fermi devices only	No
Recursion support	Yes, Fermi devices only	No
Memory buffer flexibility	Limited	Extensive, especially with 1.1
Documentation available	Extensive	Moderate
Newest release version	3.1	1.1
Printf support	Yes, Fermi devices only	Yes, but depends on device and requires extension

Table 9-2 CUDA versus OpenCL feature comparison.

*Requires existing implementation on this platform, but these exist for all three operating systems.

10. Next Steps

We first propose that we conduct more interviews with code teams and with key LC staff. The purpose of these interviews is to learn and document what these teams and staff will need to ensure a successful migration to exascale-suited programming models. ***After completing the interviews, we will propose additional deliverables that address these needs.*** It is critical that we ask the right questions in these interviews. By determining how our interviewees currently make long-term decisions about their domain of expertise and then implement those decisions, we can create deliverables that directly aid them in their decision-making and implementation process.

We propose that we prepare a comprehensive survey of novel models of parallelism that could be used for exascale computing. In order to prepare this survey, we expect to do online research, talk and meet with key industry players, and talk and meet with staff at other DOE labs. This survey will include:

- Characteristics of the models
- Their anticipated suitability to several types of LLNL scientific applications
- An examination of the present state of each of these models
- Our prediction for the state of these models in an exascale timeframe
- A qualitative risk assessment of using each model
- Recommendations for steps to take to mitigate risk with each model
- Recommendations on overall approach to take
- Implementation timeline with milestones

Similar to the deliverables that will come out of our interview process, this survey will be a useful tool for both application teams and LC staff in preparing for these future models, planning future resource allocation, planning for migration to the models, and making decisions about what actions to take now.

We propose porting an MPI application coded in C that has not already been parallelized with OpenMP to OpenMP, CUDA, and/or OpenCL. Depending on the progress of our survey, we may also port the code to one or more additional to-be-identified models. Performance of the ported application will be a primary study result. We will also track and report on the time spent porting to each target coding model. This finding will constitute a significant study result.

We do not propose moving forward with studying UPC as an alternative to OpenMP for intranode parallelism. We may identify a code such as LEOS that would benefit from a design shift to using global arrays. If we do, this code would be a candidate for porting to UPC, and we may incorporate it into our study. Our hypothesis is that porting to UPC/MPI would take considerably more time than porting to OpenCL/MPI, CUDA/MPI, or OpenMP/MPI. This is because porting to UPC will require an examination and possible recode of every single function, whereas porting to CUDA, OpenCL, and OpenMP will only require modification to the routines that are to be parallelized.

CUDA and OpenCL are fairly similar, with CUDA having more flexibility in language constructs and better support, while OpenCL is an open standard with more flexibility in target architectures. In November PGI will be releasing a CUDA C compiler that will run CUDA code on an X86-64 CPU. Intel has stated that they will release a version of OpenCL that runs on Intel CPUs late in calendar year 2010. Once this is released, it will permit us to directly compare OpenCL to CUDA and OpenMP on a specific CPU architecture as opposed to a GPU-only architecture. CUDA support is much more robust than OpenCL support. Allinea DDT and Totalview support debugging of CUDA applications. This would be a major

reason to prefer CUDA over OpenCL for a large application port. However, CUDA is an NVIDIA product. OpenCL is an open standard language and for this reason should not be ruled out.

***To stay current on the differences between CUDA and OpenCL, we propose devoting effort to an ongoing analysis of feature improvement in both languages*.** The deliverable associated with this will be a feature comparison between CUDA and OpenCL , with the anticipated audience being application developers.

***We will be able to prepare a timeline and resource requirements for the above deliverables once we confirm interest in each of them from our stakeholders*.**

Appendix A. UPC Translator Code Insertion

The UPC translator works by translating UPC code into standard C and inserting it in place of the UPC code so that it can be compiled with a normal C compiler. A call to the function below (or a similar function depending on the specifics of the variable type) is inserted by the Berkeley UPC translator once for every shared pointer modification. A similar call is inserted for every shared pointer access independent of modification.

Note that if the strict memory model is in use, the *isstrict* variable below will evaluate to nonzero, and a number of additional functions are called as a result.

```
__attribute__((__always_inline__)) static inline
void
_upcr_put_pshared(upcr_pshared_ptr_t dest, ptrdiff_t destoffset, const void *src,
size_t nbytes, int isstrict )
{

   static char _bupc_dummy_PASS_GAS = (char)sizeof(_bupc_dummy_PASS_GAS);

   upcri_local_t local = upcri_thread2local[upcr_threadof_pshared(dest)];

   (dest);

   ((void)0);

   if (local) do {
      {
         if (isstrict) gasneti_local_wmb();
         ((void)0);
         do {
            switch(nbytes) {
               case 0: break;
               case 1: *((gasnete_anytype8_t *)(((void *)(local + upcr_addrfield_pshared(dest) + destoffset)))) =
                           *((gasnete_anytype8_t *)(src));
               break;
               case 2: *((gasnete_anytype16_t *)(((void *)(local + upcr_addrfield_pshared(dest) + destoffset)))) =
                           *((gasnete_anytype16_t *)(src));
               break;
               case 4: *((gasnete_anytype32_t *)(((void *)(local + upcr_addrfield_pshared(dest) + destoffset)))) =
                           *((gasnete_anytype32_t *)(src));
               break;
               case 8: *((gasnete_anytype64_t *)(((void *)(local + upcr_addrfield_pshared(dest) + destoffset)))) =
                           *((gasnete_anytype64_t *)(src));
               break;
               default: memcpy(((void *)(local + upcr_addrfield_pshared(dest) + destoffset)), src, nbytes);

            }

         }
         while(0);
         if (isstrict) _gasneti_local_mb();

      }
      ;

   }
   while (0);
   else do {
      {
         if (isstrict) gasneti_local_wmb();
         _gasnet_put(upcri_pshared_nodeof(dest),_upcri_pshared_to_remote_off(dest, destoffset ),(void *)src,nbytes );
         if (isstrict) _gasneti_local_rmb();
```

```
      }
      ;

   }
   while (0);


}
```

Appendix B. Exascale Computing Programming Models

Two programming models are currently being proposed for exascale computing. The first is the hybrid model, in which MPI is used for inter-node programming and something else for intranode programming. The second is the unified model, in which a single notation is used for both inter- and intranode programming. For our study, the CLOMP code represented the hybrid model, and LEOS represented the unified model. The two are explained more below.

1. **Hybrid/evolutionary: MPI + _____?**

   Intranode options

   - OpenMP
     - would require extensions to support accelerator programming
     - e.g., similar to directives from PGI, CAPS
     - may require the introduction of locality-oriented concepts
     - these efforts are already underway as part of OpenMP 3.0
   - PGAS languages
     - already support a notion of locality in a shared namespace
     - UPC/CAF would need to relax strictly SPMD execution model
   - Sequoia: supports a strong notion of vertical locality
   - CUDA/OpenCL: Could be a lower level than ideal for an end user

2. **Unified/holistic: _____?**

   (a single notation for inter- and intra-node programming)

   - traditional PGAS languages: UPC, CAF, Titanium
     - would likely require extensions to handle nested parallelism,
     - vertical locality
   - HPCS languages: Chapel, X10, Fortress(?)
     - designed with locality and post-SPMD parallelism in mind
     - other candidates: Charm++, Global Arrays, ParalleX,

Appendix C. Links, Tutorials, Places to Go to Learn More

Throughout our interdependent journeys to learn about UPC, CUDA ad OpenCL, we kept track of the sources for learning that are available via books and on-line. Below are collections of our sources.

C.i. UPC Learning Experience

Research for UPC is being done at a number of different Academic sites, with different spins:

UPC@GWU

The UPC working group at the High Performance Computing Lab (HPCL) , George Washington University (GWU) is involved in a number of efforts: UPC specification, UPC testing strategies, UPC documentation, testing suites, UPC benchmarking, and UPC collective and Parallel I/O specification.

- **Berkeley UPC**

  The goal of the UPC effort at LBL and UC Berkely is to build portable, high performance implementations of UPC for large-scale multiprocessors, PC clusters, and clusters of shared memory multiprocessors. There are three major components to this effort: lightweight communication, compilation techniques for explicitly parallel languages, application benchmarks.

- **UPC@MTU**

  Michigan Tech University (MTU) projects include the recent release of the MuPC run time system for UPC as well as collective specification development, memory model research, programmability studies, and test suite development.

- **GCC UPC**

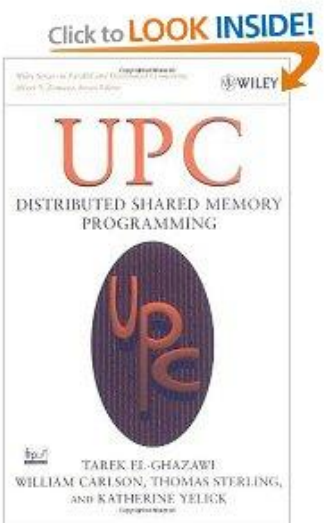
  The GCC UPC toolset provides a compilation and execution environment for programs written in the UPC. The GCC UPC compiler extends the capabilities of the GNU GCC compiler. The GCC UPC compiler is implemented as a C Language dialect translator, in a fashion similar to the implementation of the GNU Objective C compiler.

- **UPC@Florida**

  Researchers at the University of Florida are currently involved in the research and development of a next-generation performance analysis tool supporting UPC. This tool will facilitate users in identifying bottlenecks in their programs and will serve as a testbed for advanced analysis techniques aimed at increasing programmer productivity.

---

1. Book: ***UPC: Distributed Shared-Memory Programming (Hardcover)***
   ISBN-13 978 0-471-22048-0 (cloth)
   ISBN-10 0-471-22048-5 (cloth)



   Tarek El-Ghazawi (Author), William Carlson (Author), Thomas Sterling (Author), Katherine Yelick (Author) Price: **$120.00**

2. http://upc.lbl.gov/- Berkeley UPC - Unified Parallel C - *(A joint project of LBNL and UC Berkeley)* –
   a. This web site talks about the project from the UCB/LBNL perspective, and gave me names of people at LBNL/UCB to begin to pester, like Paul Hargrove.
   b. The *Downloads* tab is where I got the downloadable version (plus the README files to tell me what to do) that I am built and am currently running on Yana, and it is where I figured out about the translator
   c. I did not find this to be a helpful website – I probably should have joined their user groups – that might have helped.
   d. Under *Publications* tab, there are a few posters and talks that were initially helpful, however, I thought the book did a better job.
3. http://upc.lbl.gov/docs/system/index.html
   a. I cannot remember how I got to this documentation – probably by pestering Paul Hargrove – this was useful in describing the layers of UPC – and how they fit together.
4. http://upc.gwu.edu/
   a. This is the UPC web site at George Washington University –
      i. I like this website a little better then the UCBerkeley website.
      ii. http://upc.gwu.edu/documentation.html
         1. UPC Language Specification (V 1.2)
         2. UPC Manual – used the book more than this manual
      iii. http://www.upc.mtu.edu/tutorials.html - going onto the Michigan Tech website, you will find some example problems.
      iv. http://www.gwu.edu/~upc/download.html - Testing Suites - I have yet to try these on the compiler I built – it would certainly be worth it to try them.

1. The testing suite is intended to test the functionality of any implementation of the UPC compiler and allow the user to measure the degree of its conformance to the UPC standard. The suite should contain a set of portable test programs. These programs fall under either of the following categories:
    a. Positive tests: These tests are to verify that UPC features work properly according to the syntax and semantics described in the UPC specifications.
    b. Negative tests: These tests are to determine the error detection capabilities of a UPC compiler implementation.
        i. GWU Unified Testing Suite(GUTS), September 2008
        ii. Unified UPC Test Suite 1.2.0-r1, June 2005
        iii. The GWU Testing Strategy 1.1, March 2003
        iv. The GWU Testing Suite 1.1, September 2004
        v. The GWU UPC-IO Testing Strategy 1.2.0-r1, June 2005
        vi. The GWU UPC-IO Test Suite 1.2.0-r1, June 2005
        vii. MuPC Test Suite, January 2003

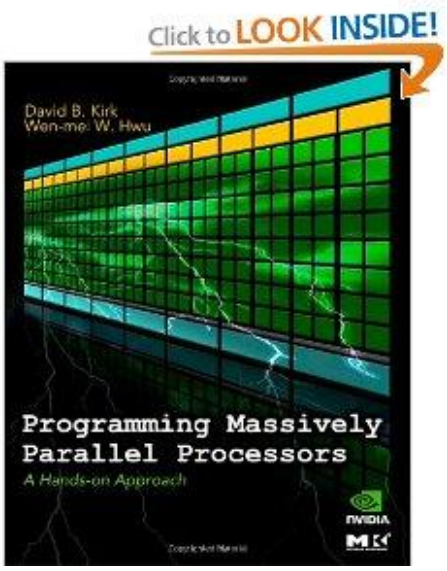
5. UPC Articles worth looking into:
    a. Hybrid Parallel Programming with MPI and Unified Parallel C, about to be published, James Dinan, P. Sadayappan (Ohio State); Pavan Balaji, Ewing Lusk, Rajeev Thakur (Argonne). Excellent paper, plus corresponding with authors. First real hybrid of MPI+UPC application with good results. *mpiupc_cf10.pdf*
    b. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures, published Euro PVM/MPI 2009, Dami´an A. Mall´on[1], Guillermo L. Taboada[2], Carlos Teijeiro[2], Juan Touri˜no, Basilio B. Fraguela[2], Andr´es G´omez[1], Ram´on Doallo[2], and J. Carlos Mouri˜no. Excellent recent article on timings between these different approaches. *Recent advances in parallel.pdf*
    c. Execution Model of three parallel languages: OpenMP, UPC, CAF, published ISO press 2005, Arni Marowka. Good article describing these three approaches, and the pros and cons. *Scien001.PDF*
    d. Unified Parallel C - UPC on HPCx, Ian Kirker and Adrian Jackson, January 14, 2008, HPCx Capability Computing. This document outlines the basic concepts of UPC, and explores what functionality is available on HPCx. It then goes on to analyze the performance of UPC against IBM's MPI and LAPI on HPCx. Both IBM's UPC offering, and an open-source (Berkeley) UPC compiler are evaluated. *HPCxTR0709.pdf*
    e. http://www.cug.org/1-conferences/CUG2010/pages/1-program/final_program/CUG10CD/CUG10_Proceedings/pages/authors/06-10Tuesday/9B-Alam-slides.pdf - Evaluation of Productivity and Performance Characteristics of CCE, CAF and UPC Compilers, by Sadaf Alam, William Sawyer, Tim Stitt, Neil Stringfellow, and Adrian Tineo. Excellent, current article given at CUG 2010.
    f. http://www.prace-project.eu/documents/13_pgas_sa.pdf - Productivity Analysis of Integrated Compilers for PGAS Languages by Sadaf Alam at the PRACE (Partnership for

Advanced Computing in Europe) Workshop "New Languages & Future Technology Prototypes", March 1-2, 2010.

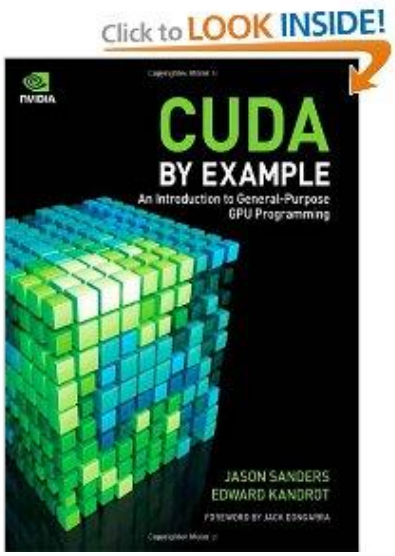
C.ii. CUDA Learning Experience

Currently, two books exist – the first as a textbook, and the other has a good sampling of examples:

1. Programming Massively Parallel Processors: A Hands-on Approach, by David B. Kirk and Wen-mei W. Hwu



2. Cuda by Example by by Jason Sanders and Edward Kandrot



CUDA has online courses to help you get started programming or teaching CUDA as well as links to Universities teaching CUDA.

For a beginner, start with *Introductory CUDA Technical Training Courses* which give an overview of the CUDA programming model and basic concepts. Try to register for one of Nvidia's CUDA webinars as well. Although there are previously recorded webinars available on Nvidia's website, live webinars are a good place to ask someone at Nvidia questions.

http://developer.nvidia.com/object/cuda_training.html
CUDA University is organized into three sections to get you started
Introductory CUDA Technical Training Courses
CUDA University Courses
CUDA Seminars and Tutorials

C.ii.a. Introductory CUDA Technical Training Courses

- Volume I: Introduction to CUDA Programming (94 pages long)
  - Exercises (for Linux and Mac) (Tar file)
  - Visual Studio Exercises (for Windows)
  - Instructions for Exercises (12 pages long)
- Volume II: CUDA Case Studies (Real examples with coding and performance, etc.)
  - Computational Finance in CUDA.....1
    1. Black-Scholes pricing for European options 3
    2. MonteCarlo simulation for European options.....16
  - Spectral Poisson Equation Solver 40
  - Parallel Reduction.....63

C.ii.b. CUDAcasts - Downloadable CUDA Training Podcasts

- Introduction to GPU Computing
- CUDA Programming Model Overview
- CUDA Programming Basics - Part I
- CUDA Programming Basics - Part II

Additional GPU Computing Online Seminars

- Introduction to MainConcept's CUDA H.264/AVC Encoder
- Monitoring and Managing GPU Clusters with Bright Cluster Management
- An Introduction to OpenCL™ Application Development with gDEBugger CL
- Rapid Prototyping and Visualization with OpenCL Studio
- GPU Computing using CUDA C – An Introduction
- GPU Computing using CUDA C – Advanced 1
- GPU Computing using CUDA C - Advanced 2
- GPU Computing using OpenCL- An Introduction
- GPU Computing using OpenCL Advanced 1
- Parallel Nsight - An Introduction and Overview
- Thrust, A C++ Standard Template Library for CUDA C - An Introduction

C.ii.c. CUDA University Courses

University of Illinois : ECE 498AL

Taught by Professor Wen-mei W. Hwu and David Kirk, NVIDIA Chief Scientist.

- Introduction to GPU Computing (60.2 MB)
- CUDA Programming Model (75.3 MB)
- CUDA API (32.4 MB)
- Simple Matrix Multiplication in CUDA (46.0 MB)
- CUDA Memory Model (109 MB)
- Shared Memory Matrix Multiplication (81.4 MB)
- Additional CUDA API Features (22.4 MB)
- Useful Information on CUDA Tools (15.7 MB)
- Threading Hardware (140 MB)
- Memory Hardware (85.8 MB)
- Memory Bank Conflicts (115 MB)
- Parallel Thread Execution (32.6 MB)
- Control Flow (96.6 MB)
- Precision (137 MB)

These classes are each downloadable CUDAcasts with video pre-scaled to be compatible with major players.

All PowerPoint class presentations can be found on the course syllabus: ECE 498AL

Stanford University: CS193G
Taught by Jared Hoberock and David Tarjan

- Introduction to Massively Parallel Computing
- GPU History and CUDA Programming Basics
- CUDA Treads and Atomics
- CUDA Memories
- Performance Considerations
- Parallel Patterns I
- Parallel Patterns II
- Introduction to Thrust
- Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors
- PDE Solvers
- The Fermi Architecture
- Ray Tracing Case Study
- Future of Throughput
- Path Planning Case Study
- Optimizing GPU Performance
- Final lecture TBD

PowerPoint versions of these presentations can be found here.
CS193G Assignments
CS193G Tutorials

UC Davis: EE171, Parallel Computer Architecture

Taught by John Owens, Associate Professor

- Course Materials

Universities teaching CUDA where you can apply to enroll or register for courses.

C.ii.d. CUDA Seminars and Tutorials

- GPU Technology Conference: search for recordings via the interactive session calendar
- SC09
  - NVIDIA GPU Computing Theatre
  - SC09 Tutorial: High Performance Computing with CUDA
- SC08 Tutorial: High Performance Computing with CUDA
- SC07 Tutorial: High Performance Computing with CUDA
- NVISION 08 Tutorials
  - Getting Started with CUDA (covers CUDA programming model, basics of CUDA programming, and BLAS and FFT libraries)
  - Advanced CUDA Training (covers 10-series architecture and optimization techniques using particle simulation and finite difference case studies)
  - All presentations from NVISION 08
- ISC 2008 Case Study: Computational Fluid Dynamics (CFD)

C.ii.e. CUDA Consultants and Trainings

- Acceleware Professional Services
- Stone Ridge Technology
- Wipro Global Consultancy Services

Dr. Dobb's Article Series

- CUDA, Supercomputing for the Masses: Part 1
  CUDA lets you work with familiar programming concepts while developing software that can run on a GPU
- CUDA, Supercomputing for the Masses: Part 2
  A first kernel
- CUDA, Supercomputing for the Masses: Part 3
  Error handling and global memory performance limitations
- CUDA, Supercomputing for the Masses: Part 4
  Understanding and using shared memory (1)
- CUDA, Supercomputing for the Masses: Part 5
  Understanding and using shared memory (2)

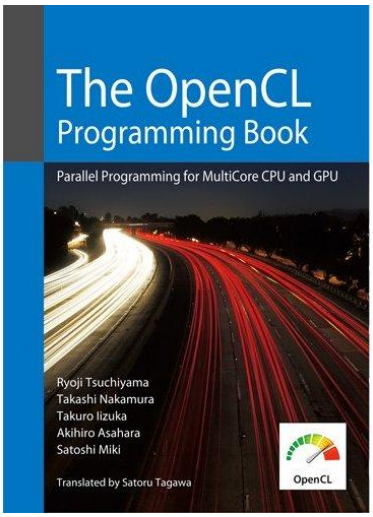
- CUDA, Supercomputing for the Masses: Part 6
  Global memory and the CUDA profiler
- CUDA, Supercomputing for the Masses: Part 7
  Double the fun with next-generation CUDA hardware
- CUDA, Supercomputing for the Masses: Part 8
  Using libraries with CUDA
- CUDA, Supercomputing for the Masses: Part 9
  Extending High-level Languages with CUDA
- CUDA, Supercomputing for the Masses: Part 10
  CUDPP, a powerful data-parallel CUDA library
- CUDA, Supercomputing for the Masses: Part 11
  Revisiting CUDA memory spaces
- CUDA, Supercomputing for the Masses: Part 12
  CUDA 2.2 changes the data movement paradigm
- CUDA, Supercomputing for the Masses: Part 13
  Using texture memory in CUDA
- CUDA, Supercomputing for the Masses: Part 14
  Debugging CUDA and using CUDA-GDB
- CUDA, Supercomputing for the Masses: Part 15
  Using Pixel Buffer Objects with CUDA and OpenGL

C.iii. OpenCL Learning Experience

Excellent GPU Computing web site: http://www.gpucomputing.net/
OpenCL tutorial from this site: http://www.gpucomputing.net/?q=node/128

- Video tutorial is especially useful
- Author has extensive CUDA experience and is able to compare and contrast OpenCL with CUDA
- Slides are also available

There is also an OpenCL book that has been published:
http://www.fixstars.com/en/company/books/opencl/
The OpenCL Programming Book, By Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, and Akihiro Asahara



- Amazon review indicates that the majority of information from this can be found on the web, however

A C++ oriented tutorial is available from AMD's developer site:
http://developer.amd.com/gpu/ATIStreamSDK/pages/TutorialOpenCL.aspx

- Not very thorough and C++ only

GPU Technology Conference 2009 Materials
http://developer.download.nvidia.com/compute/cuda/docs/GTC09Materials.htm
Specifically, OpenCL introductory material:
Slides: http://www.nvidia.com/content/GTC/documents/1409_GTC09.pdf
Video: http://www.nvidia.com/content/GTC/videos/GTC09-1409.mp4

- Good overview of Khronos group and OpenCL
- A bit NVIDIA-heavy since the people presenting work there

MacResearch Tutorials
http://www.macresearch.org/opencl

- Somewhat Apple-specific information, but good, thorough video tutorials

Slides from PPAM 2009 Tutorial
http://gpgpu.org/ppam2009
http://gpgpu.org/wp/wp-content/uploads/2009/09/

- Scientific computing emphasis
- "Clusters" portion of the presentation especially applicable to science codes

Parallel Programming Tutorials Series, Part 9
http://www.multicoreinfo.com/2009/08/parprog-part-9/

- Links to other resources
- Includes tutorials for other technologies (pthreads, MPI, MapReduce, etc.)
- Somewhat dated material (2008-2009)

Apple Developer Resources (Reference Library)

http://developer.apple.com/mac/library/documentation/Performance/Conceptual/OpenCL_MacProgGuide/Introduction/Introduction.html

- Somewhat basic material, but good introduction

NVIDIA GPU Computing Resources
http://developer.nvidia.com/object/gpu_computing_online.html

- Up-to-date and relevant material
- Missed OpenCL webinars this past week ☹

AMD Stream SDK/OpenCL Resources
http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx

- Lots of development examples and tutorials
- Good documentation
- Video Tutorials: http://developer.amd.com/documentation/videos/OpenCLTechnicalOverviewVideoSeries/Pages/default.aspx

Khronos Group OpenCL Web Site
http://www.khronos.org/registry/cl/

- Official API registry, has header files and API docs

Khronos Group YouTube Videos (SIGGRAPH 2010)
http://www.youtube.com/user/khronosgroup

Supercomputing 2009 Tutorial
www.multicoreinfo.com/2009/08/parprog-part-9

- Very thorough tutorial by some of the best names in industry
- Includes real application examples from real codes

ENJ Tutorials
http://enja.org/

- Somewhat beginner in nature but source code provided

SIGGRAPH Asia 2009 Tutorial
http://sa09.idav.ucdavis.edu/

- Good introductory tutorial

DOE Talks
Petascale computing on Sequoia
https://hpcrd.lbl.gov/scidac09/talks/Seager-Sequoia4SciDACv1.pdf

NVIDIA-Specific Notes (from CUDA Toolkit 3.1)

- Make sure that the nvidia-specific CUDA toolkit path variables are set
- MUST have a *clGetPlatformIDs* call before getting device(s)
- MUST specify platform in context setup
    - From NVIDIA OpenCL Implementation Notes 3.1:
    - "clGetPlatformInfo and/or clGetDeviceIDs will fail with the CL_INVALID_PLATFORM error if platform is NULL."

Double-Precision Floating Point Support

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable // use 64-bit fp
```

- included in **kernel** (.cl) code