

# Optimization for the Cray XE6 Interlagos Architecture

ERDC Scientific Computing Research Center  
Monika Jankun-Kelly

---

<b>1. Optimizing Code and Core Affinity for the Cray XE6 Interlagos Architecture</b>	<b>2</b>
1.1. Optimizing Core Affinity	2
1.2. Optimizing Code	2
<b>2. Interlagos Architecture</b>	<b>3</b>
2.1. Compute Node Layout	3
2.2. Memory Hierarchy	5
2.2.1. Cache Utilization and Performance	6
<b>3. Optimizing Core Affinity</b>	<b>7</b>
3.1. Single Stream Mode/Dual Stream Mode	8
3.1.1. Compiling Code for Single/Dual Stream Mode	9
3.1.2. Running Code in Single/Dual Stream Mode	10
3.1.3. Single Stream Hybrid/Threaded Codes	15
3.2. More Memory Per Process	19
3.3. Core Specialization	21
<b>4. Optimizing Code</b>	<b>23</b>
4.1. Loop Unrolling	23
4.1.1. Compiler Optimization Level	23
4.1.2. Cache Blocking	24
4.1.3. Cache Blocking Example	25
4.1.4. Vectorized Math Operations	27
4.1.5. Loop Unrolling Example	28
4.2. Coding Guidelines for Loop Unrolling	29
4.2.1. Nested Loops, Multidimensional Arrays, and Data Access	29
4.2.2. Inline Function Calls Within Loops	29
4.2.3. Avoid Dependencies	30
4.2.4. Avoid Pointer Aliasing	31
4.2.5. Align Data to Vector Boundary or Cache Line	31
4.3. Fused Multiply Add (FMA)	32
4.3.1. Compiler Optimization Options	32
4.3.2. Pitfalls to Avoid	32
4.4. Dynamic Memory Allocation	33
4.4.1. Memory Allocation on the Cray XE6	33
4.4.2. MPI Memory Allocation Example	33
4.4.3. Threaded Codes	34
4.4.4. Shared Memory Codes	34
<b>5. Further Reading</b>	<b>35</b>

## **1. Optimizing Code and Core Affinity for the Cray XE6 Interlagos Architecture**

The techniques listed below can be used to optimize the performance of your code on garnet and copper, which have Interlagos processors. Optimizing for Interlagos may not require rewriting code. For example, core affinity, the manner in which processes are assigned to cores, can be optimized quickly and simply, from the aprun command line. Some code optimization can be achieved simply by changing compiler flags. Even greater performance improvement may come from altering the code. All optimization techniques below can be applied better after gaining a basic understanding of the organization of cores, cache, and memory in the Interlagos architecture, described in the next section.

### **1.1. Optimizing Core Affinity**

Optimizing core affinity, the assignment of processes and threads to cores, requires little or no rewriting of code. The following methods may be used:

- single stream/dual stream mode
- more memory per process
- core specialization

### **1.2. Optimizing Code**

Code optimization may require some code rewriting (or it may simply require changing compiler flags). Code optimization can improve code performance in ways not possible by simply optimizing core affinity. The following methods may be used:

- loop unrolling
  - cache blocking
  - vectorized math operations
- fused multiply add
- "first touch" dynamic memory allocation

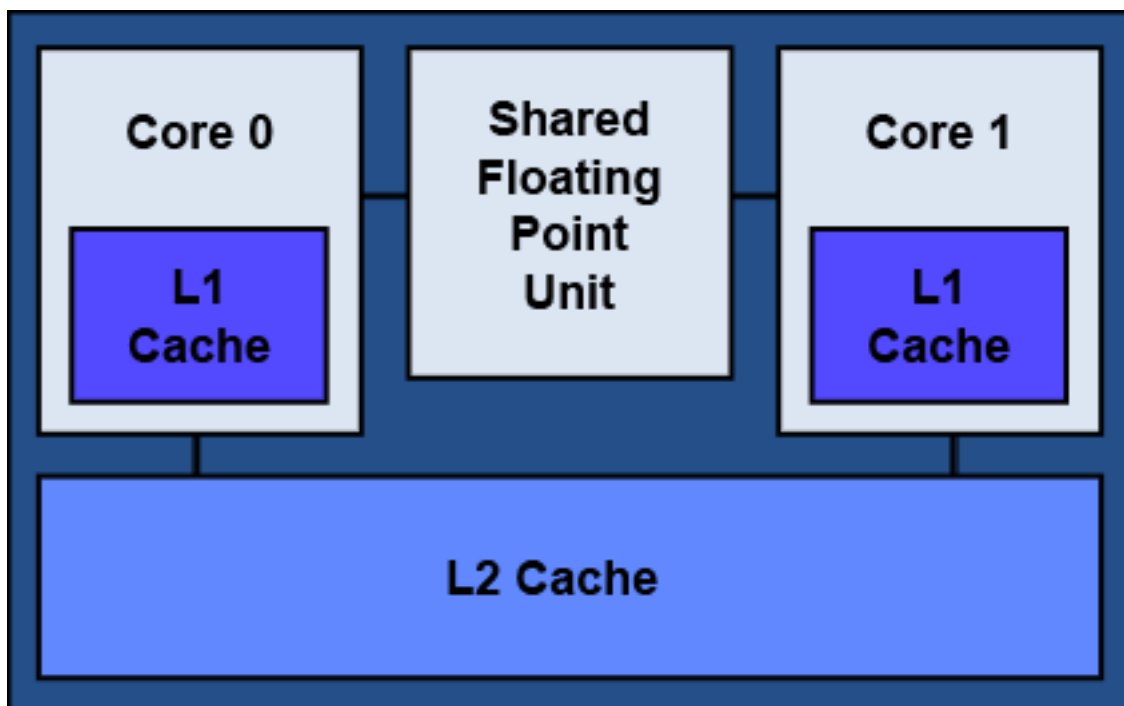
## 2. Interlagos Architecture

To achieve optimal performance from your code, it is helpful to understand the basics of the Interlagos architecture. Extensive, detailed knowledge is not required. By understanding how cores are grouped and connected, and the number of cores that share certain resources, even novice users can apply some simple yet powerful optimization techniques, which are discussed later.

### 2.1. Compute Node Layout

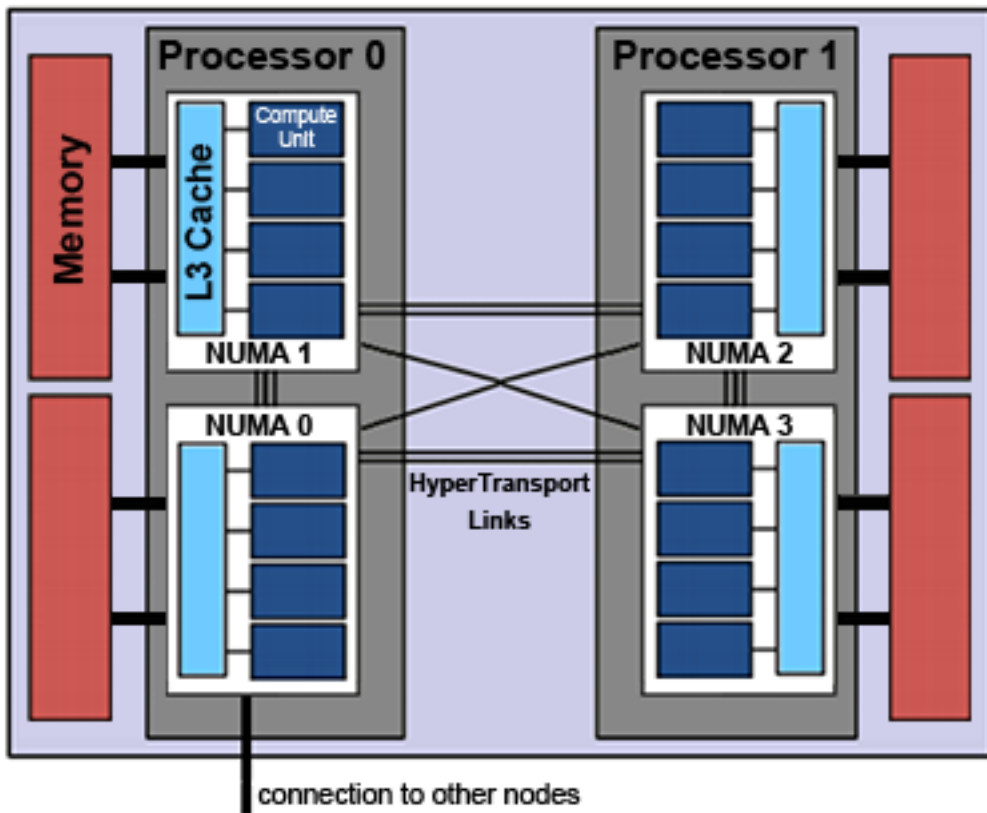
Each compute node consists of 32 cores organized into pairs, called "compute units." The two integer cores in a compute unit each have their own L1 cache, but share an L2 cache and a floating point unit (FPU). The diagram below illustrates a single compute unit (core pair).

#### Compute Unit



Compute units are grouped into NUMAs (non-uniform memory access). Each NUMA contains 4 compute units (a total of 8 cores), an L3 cache, and 16 GBytes of memory. There are four NUMAs (two processors) on each compute node as shown in the top section of the diagram below, and a total of 64 GBytes of memory. The four NUMAs on a node are connected to each other by HyperTransport links, which allow a core on one NUMA to access the memory on the other NUMAs. Thus, every core on a compute node can access all 64 GBytes of memory on the node.

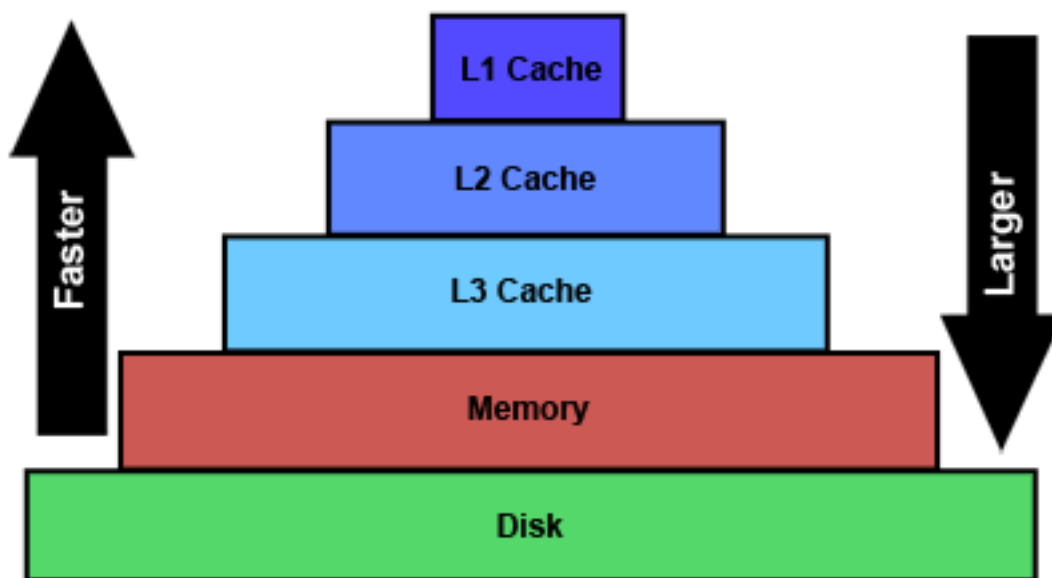
## Compute Node



## 2.2. Memory Hierarchy

The memory hierarchy balances the competing needs for fast data access and large data capacity. The memory hierarchy is made up of levels of increasing size and increasing access time, as shown in the lower half of the diagram below and the table on the following page. A core's L1 cache has the fastest access time, followed by the L2 cache of a core pair, then the L3 cache of a NUMA (8 cores), then the NUMA's memory, then processor's memory, and finally the node's memory. A core can most quickly access memory on its own NUMA (8 cores). Accessing memory on a neighboring NUMA requires going through HyperTransport links, which is slower. Memory on a core's own processor can be accessed more quickly than memory on the other processor. The two NUMAs on a processor (16 cores) share memory address space. Accessing memory on another node requires using the Gemini network and is much slower than accessing memory on a core's own node. Accessing the disk is extremely slow. The table below shows that access time for cache, memory, another node, and disk differs by several orders of magnitude.

### Memory Hierarchy



## Memory Hierarchy, Size, and Access Time

Memory Hierarchy Level	Number of Cores	Size	Access Time
L1 cache	1	16 KBytes	4 CPU cycles
L2 cache	2	2 MBytes	20 CPU cycles
L3 cache	8	6 MBytes *	45 CPU cycles
Memory (own NUMA)	8	16 GBytes	X hundred CPU cycles
Memory (other NUMA)	16	32 GBytes	2X hundred CPU cycles
Memory (other processor)	32	62 GBytes **	3X hundred CPU cycles
MPI Message to Other Node	---	---	~3000 CPU cycles or ~1.5 $\mu$ s
Disk	---	5 PBytes	millions of CPU cycles or several ms

(\*) The L3 cache size is 8 MBytes, but 2 MBytes are reserved for cache coherency, thus only 6 MBytes are accessible to the user.

(\*\*) A compute node has 64 GBytes of memory, but only 62 GBytes are accessible to the user.

### 2.2.1. Cache Utilization and Performance

The data needed for computation are either read from files on disk or created in memory. Computation is performed on data in a core's L1 cache. Therefore, data from lower levels of the memory hierarchy (disk, memory) must move up through the hierarchy to the L1 cache before it can be used in computation. The L1 cache is very small, thus as new data are brought in and computation results are generated, the L1 cache fills up. To make room in the L1 cache, older data are overwritten, and computation results are moved out of L1 cache and down the hierarchy into memory. Frequently moving data to and from the lower, slower levels of the memory hierarchy degrades performance because the core sits idle waiting for needed data to arrive in the L1 cache. Therefore it's good to maximize cache hits (needed data found in cache) and minimize cache misses (needed data not found in cache). Optimal cache utilization has a high cache hit/miss ratio. Techniques for improving cache utilization are discussed in a later section.

### 3. Optimizing Core Affinity

Core affinity is the way processes are assigned to cores. Core affinity is controlled by various `aprun` command options, shown in the table below. Changing your core affinity does not require rewriting your code. However, some core affinity optimization techniques make more resources available to each process. For greater performance improvement, you may wish to rewrite your code to take advantage of this.

<b>aprun option</b>	<b>purpose</b>
-n procs	Total number of processes
-N procs	Number of processes per node For hybrid codes, number of MPI processes per node
-S procs	Number of processes per NUMA
-d cores	Number of cores per process For hybrid codes, number of threads per process
-j 1	Run in single stream mode (The default is dual stream)

### 3.1. Single Stream Mode/Dual Stream Mode

Interlagos cores are arranged into compute units comprised of an integer core pair, a shared floating point unit (FPU), and a shared L2 cache. Understanding when to run code on both cores in a pair (dual stream mode) and when to run on just one core in a pair (single stream mode) may improve performance. In single stream mode, one core per pair remains idle, freeing some resources for exclusive use by the other active core. The possible benefits of single stream mode are listed below.

- no contention for shared floating point unit (FPU)
- entire L2 cache available to one core
- contention for L3 cache may be halved
- twice as much memory available to each core
- greater memory bandwidth
- less contention for Gemini interconnect to remote nodes

Keep in mind that some codes may run better in dual stream mode than single stream. For example, if a code's performance is not bound by FPU contention, reducing FPU contention will probably not significantly improve performance. If performance is memory bound, improving cache utilization won't help much. There are possible advantages to running in dual stream mode.

- shared memory codes, more threads can access memory on the same node
- communication between processes/threads is faster on the same node than between separate nodes
- if cores used per node is maximized, less nodes needed to run same number of processes

By default, code is compiled for and run in dual stream mode; no special options are needed. To compile code for single stream mode, use the **interlagos-cu** compute unit compile target. To run code in single stream mode, use the `aprun -j 1` option to specify only one core per compute unit should be used.

How do you determine whether your code can benefit from running in single stream mode? You can quickly and easily run small, representative test cases in both modes. Optionally, you can use profiling tools to find bottlenecks in performance. Codes whose performance is bound by floating point computations, hampered by L2 cache misses, or excessive contention for the interconnect may benefit from running in single stream mode.



### 3.1.1. Compiling Code for Single/Dual Stream Mode

Cray provides two compile targets for codes built for compute nodes. The default **interlagos** target is meant for codes running in dual stream mode. The default target will also work for codes running in single stream mode. Recompiling code for single stream mode is optional, but may improve performance. Use the default **interlagos** target for dual stream mode. Use either **interlagos** or **interlagos-cu** for single stream mode. Do not run in dual stream mode if your code was built for the **interlagos-cu** target.

The **interlagos-cu** target is optimized for codes running in single stream mode. The difference between the two targets is primarily the way in which the compiler uses the shared memory subsystem hardware resources. To change from the default **interlagos** target to **interlagos-cu**, use the `-h cpu=interlagos-cu` compiler flag, or swap modules as shown below.

```
module swap craype-interlagos craype-interlagos-cu
```

Both **interlagos** and **interlagos-cu** are Cray compiler targets, so be sure the Cray programming environment module `PrgEnv-cray` is loaded.

### 3.1.2. Running Code in Single/Dual Stream Mode

The `-j` `aprun` option specifies whether code runs in single or dual stream mode. Note that `-j` is only available on garnet because it has the newer CLE 4.1 operating system, while copper has the older CLE 4.0. The default dual stream mode uses both cores in a compute unit, while single stream mode uses one core and leaves the other idle, increasing resources available to the active core. The value of `-j` is the number of cores in a compute unit, or core pair, to use for user jobs. Refer to the compute unit diagram in the architecture section, earlier in this guide.

- `-j 0` (default) use all cores in a compute unit
- `-j 1` use one core in a compute unit
- `-j N` use N cores in a compute unit  
N can be at most 2, thus `-j 0` is equivalent to `-j 2`

When running codes on copper, know whether your code is nonthreaded, e.g., pure MPI, or threaded, e.g. OpenMP or hybrid MPI/OpenMP. On copper, the `-j` option is unavailable. Thus the `-d 2` option is used to run code in single stream mode, but this only works for nonthreaded codes. Threaded codes running in single stream mode on copper must have cores manually assigned with `-cc core_list`.

```
% single stream mode (garnet, CLE 4.1)
aprun -j 1
```

```
% single stream mode (copper, CLE 4.0 non-threaded)
aprun -d 2
```



```
% single stream mode (copper, CLE 4.0 threaded)
aprun -cc core_list
```

If your code runs on garnet and is threaded or has core affinity other than one process per core, it is important to understand the distinction between `-d` and the new `-j` option. The `-j` option controls the number of cores per compute unit (core pair) to be used. The `-d` option is the number of cores assigned to each process. The `-j` option is applied first, and the `-d` option operates on the cores left after the `-j` option has been applied. Other core affinity options, such as `-N`, processes per node, and `-S`, processes per NUMA, are also applied after `-j`.

The examples below show core affinity in dual stream mode and single stream mode for a 16-process MPI job. Recall that in dual stream mode, the 8 active cores on a NUMA share the L3 cache, and each core pair shares an L2 cache and floating point unit (FPU). In single stream mode, 4 active cores per NUMA get more L3 cache, and each core gets the whole L2 cache and exclusive use of the FPU.

```
% dual stream mode
aprun -n 16
```



NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

 user process  
 idle core

```
% single stream mode (garnet, CLE 4.1)
aprun -n 16 -j 1
```

```
% single stream mode (copper, CLE 4.0)
aprun -n 16 -d 2
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

 user process  
 idle core

The following examples illustrate the effect of using the single stream `-j1` option with other options that affect core affinity, such as `-d depth`, `-S cores_per_numa`, and `-N cores_per_node`. Any examples with the `-j` option are for garnet only, not copper.

When `-j1` and `-d` are both used, `-d` skips over twice as many cores, since `-j1` removes odd numbered cores from the available core set.

```
aprun -n 4 -d 4
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

In the example above, NUMAs 0 and 1 had two active cores sharing an L3 cache. When each NUMA has only one active core, that core gets the entire L3 cache for its own use.

```
aprun -n 4 -d 4 -j 1 (garnet only)
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

When using `-S procs_per_numa`, remember that the single stream `-j1` option changes process spacing within a NUMA. In single stream mode, 4 processes at most can be placed on a NUMA, not 8.

```
aprun -n 6 -S 3
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

```
aprun -n 6 -S 3 -j 1 (garnet only)
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

When using `-N procs_per_node`, keep in mind that in single stream mode, only 16 of the 32 cores on a node are available.

This example runs 100 processes, placing 20 processes per node, and using 5 nodes. It will work in dual stream mode because 20 cores is less than the maximum of 32 cores per node. In single stream mode, half the cores are idle and `procs_per_node` must be 16 or less.

```
% 5 nodes used, 20 processes per node, dual stream  
aprun -n 100 -N 20
```

```
% INCORRECT! (garnet only)  
% Only 16 cores available per node in single stream mode  
aprun -n 100 -N 20 -j 1
```

```
% Fixed! (garnet only)  
% Now uses 10 nodes, 10 processes per node  
aprun -n 100 -N 10 -j 1
```

### 3.1.3. Single Stream Hybrid/Threaded Codes

Single stream threaded codes and single stream hybrid codes require different aprun options on copper than on garnet, because of garnet running the newer CLE 4.1 operating system, and copper running CLE 4.0.

#### ***Copper Only Examples, Single Stream Threaded/Hybrid Codes***

When running single stream threaded codes, you must manually map threads to cores with the `-cc cpu_list` option and specify the number of MPI processes per node with the `aprun -N` option if using MPI. Furthermore, you must set the OpenMP environment variable `$OMP_NUM_THREADS` to the number of threads per MPI process.

In the single stream hybrid MPI/OpenMP examples below, we only use the even-numbered cores to ensure that one core in each core pair remains idle. That means 16 out of 32 cores per node will be used. Divide 16 by the number of MPI processes to determine the max number of threads per MPI process.

```
% single stream mode, hybrid MPI/OpenMP codes          (copper)
% 8 MPI processes, 2 OpenMP threads per MPI process
export $OMP_NUM_THREADS=2
aprun -N 8 -cc 0,2:4,6:8,10:12,14:16,18:20,22:24,26:28,30
```

```
% single stream mode, hybrid MPI/OpenMP codes          (copper)
% 4 MPI processes, 4 OpenMP threads per MPI process
export $OMP_NUM_THREADS=4
aprun -N 4 -cc 0,2,4,6 : 8,10,12,14 : 16,18,20,22 : 24,26,28,30
```

```
% single stream mode, hybrid MPI/OpenMP codes          (copper)
% 2 MPI processes, 8 OpenMP threads per MPI process
export $OMP_NUM_THREADS=8
aprun -N 2 -cc 0,2,4,6,8,10,12,14 : 16,18,20,22,24,26,28,30
```

### Garnet and Copper Example, Changing from Dual to Single Stream

This example starts with a dual stream hybrid MPI/OpenMP code, then modifies aprun options to run it in single stream mode.

```
% dual stream, hybrid code
% 4 MPI processes per node
% 8 threads per MPI process
```

```
export OMP_NUM_THREADS=8
aprun -N 4 -d 8
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

MPI process/thread 0  
 other threads

In single stream mode, there are only 16 cores available rather than 32. We must reduce either the number of MPI processes per node or the number of threads per MPI process.

```
% single stream, hybrid code
% 2 MPI processes per node
% 8 threads per MPI process
```

```
export OMP_NUM_THREADS=8
aprun -N 2 -d 8 -j 1 (garnet)
```

```
export OMP_NUM_THREADS=8
aprun -N 2 -cc 0,2,4,6,8,10,12,14:16,18,20,22,24,26,28,30 (copper)
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

MPI process/thread 0  
 other threads



```
% single stream, threaded code
% 4 MPI processes per node
% 4 threads per MPI process
```

```
export OMP_NUM_THREADS=4
aprun -N 4 -d 4 -j 1 (garnet)
```

```
export OMP_NUM_THREADS=4
aprun -N 4 -cc 0,2,4,6:8,10,12,14:16,18,20,22:24,26,28,30 (copper)
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

MPI process/thread 0  
 other threads

To change PBS script from dual stream mode to single stream mode, follow the steps below. It may help to use the core affinity chart below.

Copper, CLE 4.0 instructions

1. Double the number of cores requested (PBS `ncpus`) or halve the total number of processes (aprun `-n`)
2. Specify single stream mode
  - a. nonthreaded codes (aprun `-d2`)
  - b. threaded codes (aprun `-cc core_list`)
3. Determine if processes per node (aprun `-N`) or per NUMA (aprun `-S`) should remain the same or need to be cut in half

Garnet, CLE 4.1 instructions

1. Double the number of nodes requested (PBS `select`) or halve the total number of processes (aprun `-n`)
2. Specify single stream mode (aprun `-j1`)
3. Determine if processes per node (aprun `-N`, PBS `mpiprocs`) or per NUMA (aprun `-S`) should remain the same or may need to be cut in half

Core Affinity Chart

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

Examples of hybrid MPI/OpenMP PBS scripts can be found in the [garnet PBS guide](#) and [copper PBS guide](#).



### 3.2. More Memory Per Process

By default, an MPI job runs one process per core, with all processes sharing the available memory on the node. If you need more memory per process, then your job needs to run fewer MPI processes per node. Similarly, the eight cores on a NUMA share L3 cache. To increase L3 cache available to each process, reduce processes per NUMA with the `aprun -S` option. The two cores in each compute unit share L2 cache. To give a process exclusive use of the whole L2 cache, leave one core per pair idle. This can be done either by running in single stream mode, `aprun -j1`, or by using the `aprun -d depth` option, where *depth* is 2 or more. See the single stream/dual stream section for more information about single stream mode.

In the example below, 32 MPI processes per node fully utilize all cores on a node. This `-N 32` option is the default. Each process receives 1/32 of available memory, or 2 GBytes. Core pairs contend for the L2 cache, and all eight cores on each NUMA contend for the L3 cache.

`aprun -N 32`



NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

 user process  
 idle core

By cutting the number of processes per node in half, we double the memory available to each process from 2 GBytes to 4 GBytes. By cutting the number of processes on each NUMA in half, we reduce contention for the L3 cache.

```
aprun -N 16 -S 4
```



NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

 user process  
 idle core

Each process has exclusive use of the entire L2 cache (shared by core pairs), contends for the L3 cache with only one other process, and can use 8 GBytes of memory.

```
aprun -N 8 -S 2 -d 4
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

 user process  
 idle core

### 3.3. Core Specialization

Core specialization reserves a small number of cores for Application Level Placement Scheduler (ALPS) processes such as `apinit` and `apsched`. The rest of the cores on a node can then run user job processes without interruption by ALPS processes. Core specialization eliminates the buffer flushes and cache flushes that occur during context switch from user process to ALPS process, which may improve performance but also reduce the number of cores on a node that run user processes and may decrease performance. Dual stream mode is more likely to benefit from core specialization than single stream mode.

The `aprun -r` option is used to specify the number of cores per node dedicated to system services. The default is `-r0` and means no dedicated cores; thus system service processes will interrupt user processes. When running in dual stream mode, users should reserve entire compute units (core pairs) rather than individual cores.



```
% single stream mode
% one core reserved for system services
aprun -r 1
```

```
% dual stream mode
% core pair reserved for system services
aprun -r 2
```

Keep in mind that cores reserved for ALPS processes mean less cores available for user jobs, and adjust your `aprun -n` option accordingly.

```
% dual stream mode
% 16 user processes
% core pair reserved for system services
% 18 out of 32 cores used
aprun -n 16 -r 2
```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

 user process  
 ALPS process



```
% dual stream mode
% 30 user processes, not 32!
```

```

% core pair reserved for system services
% all 32 cores used
aprun -n 30 -r 2

```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

 user process  
 ALPS process

```

% single stream mode (CLE 4.1)
% 15 user processes, not 16!
% one core reserved for system services
% 16 cores idle, all 32 cores used
aprun -n 15 -j 1 -r 1



```

```

% single stream mode (CLE 4.0)
% 15 user processes, not 16!
% one core reserved for system services
% 16 cores idle, all 32 cores used
aprun -n 15 -d 2 -r 1

```

NUMA 0								NUMA 1							
Core 0	Core 1	Core 2	Core 3	Core 4	Core 5	Core 6	Core 7	Core 8	Core 9	Core 10	Core 11	Core 12	Core 13	Core 14	Core 15
NUMA 2								NUMA 3							
Core 16	Core 17	Core 18	Core 19	Core 20	Core 21	Core 22	Core 23	Core 24	Core 25	Core 26	Core 27	Core 28	Core 29	Core 30	Core 31

 user process  
 ALPS process

## 4. Optimizing Code

The code optimization techniques and examples that follow demonstrate how to optimize your code for the Interlagos architecture. While these techniques can be applied to many architectures, this guide shows how to apply them specifically to Interlagos.

### 4.1. Loop Unrolling

Loop unrolling is an optimization technique that can be performed by the compiler, but can also be done manually. Its goals are to reduce loop overhead, improve resource utilization, and increase parallelism. Loop unrolling can be used to improve cache utilization, the amount of data currently being used that is in the cache. It can also be used to vectorize math operations, meaning that the same operation is performed on several pieces of data at once rather than on just one piece of data.

#### 4.1.1. Compiler Optimization Level

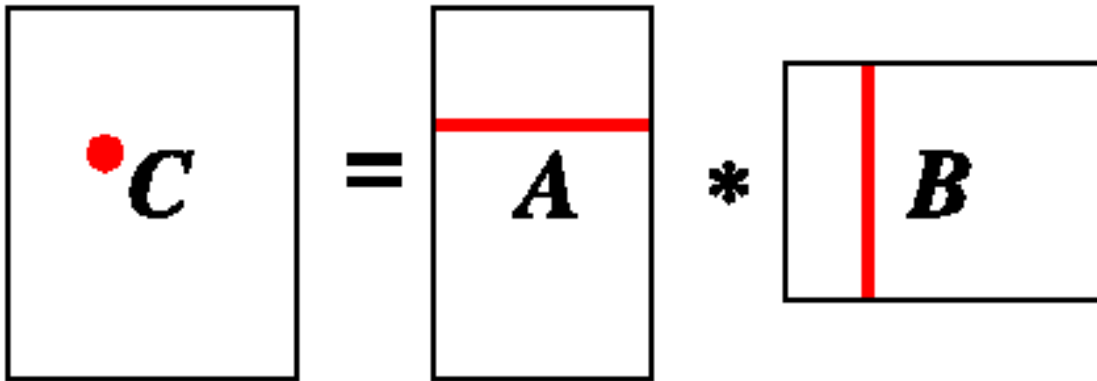
For best results, use the Cray compiler. It was designed for the Interlagos architecture, performs automatic cache blocking, and vectorizes more loops than other compilers. The GNU compiler should be the last choice. Be sure to use the appropriate compiler optimization level to enable vectorization, loop unrolling, and function inlining. For more precise control of compiler optimization, consult the compiler manual for a description of all available flags. If you wish to see feedback about optimization from the compiler, use compiler feedback flags.

Compiler	Optimization Level	Feedback Flags
Cray	-O2	-h list=a -h display_opt -rm (Fortran)
PGI	-O2	-Minfo=inform
Intel	-O2	-vec-report3
GNU	-O3 -march=bdver1	-ftree-vectorizer-verbose=2

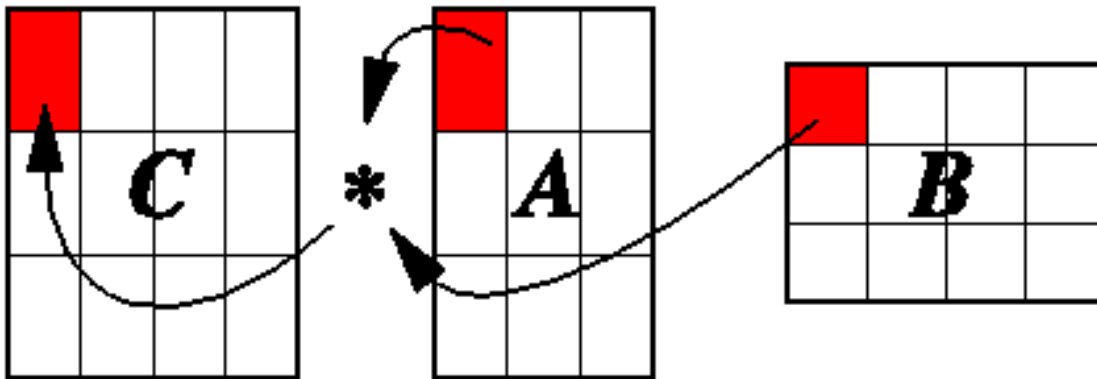
### 4.1.2. Cache Blocking

Recall that cache access is much faster than memory access. Cache blocking breaks work into blocks that fit neatly into cache, increases reuse of data in cache, and reduces the need to slowly fetch data from memory. You can have the compiler try to do this for you automatically through loop unrolling. You can also write your code in such a way that you work on data in cache-sized chunks.

Consider a simple matrix multiplication code that performs  $C = A * B$ . The original version multiplies one row of  $A$  and one row of  $B$  during each loop iteration.



A blocked matrix multiplication code multiplies a block of  $A$  by a block of  $B$  at each loop iteration.





### 4.1.3. Cache Blocking Example

This example shows how to manually optimize matrix multiplication for the garnet/copper L1 cache, and later, the L1 and L2 caches. To ensure your data fits into the cache, remember that on garnet and copper, the L1 cache is 16 KBytes, the L2 cache is 2 MBytes, and the L3 cache is 6 MBytes.

The following blocked matrix multiplication example works on data blocks that fit neatly into the L1 cache. Here is the original version. Let matrix A be of size  $M \times N$ , matrix B  $N \times P$ , and matrix C  $M \times P$ .

```
for( int row = 0; row < M; ++row )
{
    for( int col = 0; col < P; ++col )
    {
        C[row][col] = 0;
        for( int i = 0; i < N; ++i )
            C[row][col] += A[row][i] * B[i][col];
    }
}
```

Here is the manually blocked version. If profiling indicates your chosen compiler may not be performing good cache blocking, you may try to do it manually. The L1 cache is 16 KBytes. The A and B matrix blocks will be heavily reused during blocked multiplication, so we want them both in the cache. That's two blocks so each block can be 8 KBytes at most. If we wish to work on  $N \times N$ -sized blocks, that's roughly 89x89 Bytes. If our matrices hold values of type double, and each double is 4 bytes, that's roughly 22x22 doubles.

```

#define BLOCK_SIZE 22

// assume M, N, P are all evenly divisible by BLOCK_SIZE
// if not, a bit extra coding is needed, or pad the data
int row_block_count = M / BLOCK_SIZE;
int col_block_count = P / BLOCK_SIZE;

for( int row = 0; row < M; ++row )
    for( int col = 0; col < P; ++col )
        C[row][col] = 0;

for( int br = 0; br < row_block_count; ++br )
{
    for( int cr = 0; cr < col_block_count; ++cr )
    {
        int row_start = br * BLOCK_SIZE;
        int row_end   = row_start + BLOCK_SIZE;
        for( int row = row_start; row < row_end; ++row )
        {
            col_start = cr * BLOCK_SIZE;
            col_end   = col_start + BLOCK_SIZE;
            for( int col = col_start; col < col_end; ++col )
            {
                int ia, ib;
                for( int i = 0; i < BLOCK_SIZE; ++i )
                {
                    ia = i + col_start;
                    ib = i + row_start;
                    C[row][col] += A[row][ia] * B[ib][col];
                }
            }
        }
    }
}

```

If we wanted to optimize for both the L1 and L2 caches, we could do another level of nesting in the code. How many of our L1-sized matrix blocks can be fit into the L2 cache? If each block is 22x22 doubles, that's 484 doubles or 1936 Bytes. The L2 cache is 2 MBytes, so it can hold at most 1033 matrix blocks of 22x22 doubles. Recall we want to fit both A and B into the cache, so that's about 516 blocks of 22x22 doubles for A and B each. Our matrix multiplication code now looks something like this:

```

loop over entire matrix, broken into pieces of 516 blocks
    loop over 516 matrix blocks
        perform matrix multiplication on block of 22x22 doubles

```

#### 4.1.4. Vectorized Math Operations

Many science and engineering codes loop over arrays of data, performing math operations on one scalar at a time. Such codes may run faster if vectorized. Instead of working on one value of an array at a time, vectorized code works on several values at a time. The compiler will try to vectorize loops for you. You may achieve more vectorization with some code rewriting.

The Interlagos floating point unit (FPU) can perform vectorized math operations on 128 bits at a time in dual stream mode, or 256 bits in single stream mode. The table below shows the vector size, or number of simultaneous operations in a manually unrolled loop, and gives an upper bound on the speedup you may achieve from vectorization. If your loop iterates over an array of doubles, and each double is 32 bits, you can work on a vector of 8 doubles at a time in single stream mode, or a vector of 4 doubles in dual stream mode. Thus, the maximum speedup from vectorization would be 8. Please refer to an earlier section of this document for an explanation of single and dual stream modes. Integer types are shown in the table, but keep in mind that mixed floating point/integer math is done by the FPU, whereas pure integer math is performed by the integer core.

<b>array data type</b>	<b>FPU vector size (dual stream)</b>	<b>FPU vector size (single stream)</b>
byte	16	32
float	8	16
double	4	8
int	4	8
long	2	4

#### 4.1.5. Loop Unrolling Example

This simple example illustrated the concept of loop unrolling. The loop below is not unrolled. The loop counter increments by one, and one piece of data is computed in the body of the loop. If you make no change to the code below, the compiler will try to unroll it for you, provided you used the correct compiler optimization level.

```
for( int i = 0; i < data_count; ++i )
{
    result[i] = a[i] * b[i];
}
```

Here is the same loop as above, unrolled x4. The loop counter now increments by 4, and four pieces of data are computed in the body of the loop. If the compiler unrolls your loop, it will look something like this. You can also unroll the loop manually by rewriting your loop as shown below. Manual unrolling gives you greater control, but may or may not be an improvement over how the compiler would unroll loops.

```
for( int i = 0; i < data_count; i = i+4 )
{
    result[i  ] = a[i  ] * b[i  ];
    result[i+1] = a[i+1] * b[i+1];
    result[i+2] = a[i+2] * b[i+2];
    result[i+3] = a[i+3] * b[i+3];
}
```

Here is the unrolled loop x16, but in compact form. The compiler directive explicitly asks the compiler to unroll the loop, saving you some typing for longer unrolled loops. C and C++ code uses `#pragma unroll`, while Fortran uses `!DIR$ UNROLL N`, meaning unroll the loop xN.

```
for( int i = 0; i < data_count; i = i+16 )
{
    #pragma unroll
    for( int j = i; j < i+16; ++j )
        result[j] = a[j] * b[j];
}
```

## 4.2. Coding Guidelines for Loop Unrolling

Follow these coding guidelines to make your loops more suitable for unrolling by the compiler and to make your data structures more suitable for efficient vectorization and cache blocking.

### 4.2.1. Nested Loops, Multidimensional Arrays, and Data Access

Nested loops are often used to work on multidimensional arrays. Be aware of language-dependent data access patterns. C and C++ will make the most efficient use of memory and cache when accessing data in row-major order. That means for a 2D matrix, within each row, you stride across the columns. Fortran is best with column-major order. Within each column, you stride down the rows.

```
// C, C++
for( row=0; row<n; row++ )
    // unroll this column loop
    for( col=0; col<n; col++ )
        a[row][col]=b[row][col]*s;

! Fortran
do col=1,n
    ! unroll this row loop
    do row=1,n
        a(row,col)=b(row,col)*s
    enddo
enddo
```

### 4.2.2. Inline Function Calls Within Loops

Function calls in a loop can render it nonvectorizable. Inline functions when possible, either with appropriate compiler optimization levels or explicit compiler directives, such as C's `#pragma inline` or Fortran's `!DIR$ INLINE`.

```
#pragma inline
double dot( double * b, double * a )
{ return( a[0]*b[0] + a[1]*b[1] + a[2]*b[2] ); }

for( int i = 0; i < vec_count; ++i )
{
    res[i] = dot( vecsA[3*i], vecsB[3*i] );
}
```

After inlining the code looks like this and can be vectorized.

```
for( int i = 0; i < vec_count; ++i )
{
    res[i] = vecsA[i]*vecsB[i] + vecsA[i+1]*vecsB[i+1]
           + vecsA[i+2]*vecsB[i+2];
}
```

### 4.2.3. Avoid Dependencies

Loop dependence means the loop must be executed sequentially to produce the desired result. Thus the loop cannot be parallelized. If possible, try to rewrite the computation in a nondependent way.

#### *Read After Write*

This loop is not parallelizable. If unrolled, element  $a[i-1]$  would be read before its correct value had been written.

```
for( int i = 1; i < count; ++i )
    a[i] = a[i-1] + b[i];
```

initial values	sequential result	parallel result
a0 = 4 b0 = 1	(correct)	(INCORRECT)
a1 = 3 b1 = 2	a1 = 4+2 = 6	a1 = 4+2 = 6
a2 = 5 b2 = 3	a2 = 6+3 = 9	a2 = 3+3 = 6
a3 = 2 b3 = 1	a3 = 9+1 = 10	a3 = 5+1 = 6
a4 = 8 b4 = 5	a4 = 10+5 = 15	a4 = 2+5 = 7

#### *Write After Read*

This loop can be parallelized. The original and unrolled versions produce the same result because element  $a[i+1]$  does not have to be written before it is read.

```
for( int i = 0; i < count-1; ++i )
    a[i] = a[i+1] + b[i];
```

initial values	sequential result	parallel result
a0 = 4 b0 = 1	(correct)	(correct)
a1 = 3 b1 = 2	a0 = 3+1 = 4	a0 = 3+1 = 4
a2 = 5 b2 = 3	a1 = 5+2 = 7	a1 = 5+2 = 7
a3 = 2 b3 = 1	a2 = 2+3 = 5	a2 = 2+3 = 5
a4 = 8 b3 = 5	a3 = 8+5 = 13	a3 = 8+5 = 13

#### 4.2.4. Avoid Pointer Aliasing

Aliasing happens when two pointers point to the same memory location. When a loop uses pointers, it might not vectorize because the compiler assumes there could be dependencies. To prevent this, either never alias, or when you know a certain loop has pointers but is free of aliasing, suggest or even force the compiler to vectorize with compiler directives.

The IVDEP directive suggests a following loop be vectorized.

```
// C/C++
#pragma ivdep

!Fortran
!DEC$ IVDEP
```

The VECTOR ALWAYS pragma forces a loop to be vectorized, so use caution!

```
// C/C++
#pragma vector always

!Fortran
!DEC$ VECTOR ALWAYS
```

#### 4.2.5. Align Data to Vector Boundary or Cache Line

For data whose size can be neatly divided into vector or even smaller cache-line-sized chunks can be accessed more efficiently. The Cray XE6 has a cache line size of 8 bytes and a vector size of 16 bytes in dual stream mode, or 32 bytes in single stream mode.

In C code, use the memalign function to allocate memory in chunks of XX bytes.

```
memalign( XX, size );
```

In Fortran90 code, use the `-align arrayXXbyte` compiler option.

In multidimensional arrays, padding lower dimensions for alignment may be beneficial. For example, instead of an array `a[15][16][16]`, use `a[16][16][16]`.

### 4.3. Fused Multiply Add (FMA)

Some scientific codes and many graphics codes make heavy use of the expression  $a*b + c$ , known as a multiply add. The Interlagos instruction set contains a fused multiply add (FMA) instruction that combines both operations into one instruction, which is executed faster than a multiply followed by an add. Taking advantage of FMA requires little to no code rewriting, since the compiler does most of the work if it sees expressions matching  $a*b + c$ .

Below are examples of code that would benefit from use of FMA and also from vectorization (see previous section).

```
// C / C++
for( int i = 0; i < data_count; i += 4 )
{
    D[i ] = A[i ]*B[i ] + C[i ];
    D[i+1] = A[i+1]*B[i+1] + C[i+1];
    D[i+2] = A[i+2]*B[i+2] + C[i+2];
    D[i+3] = A[i+3]*B[i+3] + C[i+3];
}

! Fortran
D[1:4] = A[1:4]*B[1:4] + C[1:4]
```

#### 4.3.1. Compiler Optimization Options

The Cray and Intel compilers are aware of the Interlagos architecture FMA capability. The PGI and GNU compilers are not. Use the Cray or Intel compilers with the O2 compiler optimization option to enable vectorized fused multiply add.

#### 4.3.2. Pitfalls to Avoid

If using the Intel compiler, the `-fma` option enables fused multiply add. This option is used by default. However, be aware that the `-fp-model strict` option turns off the `-fma` option. If using `-fp-model strict`, you must manually set the `-fma` option.



## 4.4. Dynamic Memory Allocation

Ideal dynamic memory allocation assigns memory "closest" to the core that will be using that memory. To make best use of memory, we need to understand when memory is allocated and the location and size of each part of the memory hierarchy.

### 4.4.1. Memory Allocation on the Cray XE6

Cray compute node Linux has a "first touch policy" for memory allocation. Allocation does not occur when calling an allocation function (C `*alloc`, C++ `new`, Fortran `allocate`, etc.). Allocation is deferred until memory is "touched" or accessed. Allocation occurs in memory "closest" to the touching core. Thus, dynamically allocated memory will be placed closest to the core that initialized it.

Linux assumes "swap space" exists for allocations too large to fit into local memory, but there is no swap space on a Cray XE6. Over allocation or out-of-memory errors will not be apparent until memory is "touched" or accessed.

### 4.4.2. MPI Memory Allocation Example

In MPI codes, to ensure each MPI process has its data closest to its own core, allocate after `MPI_Init` has been called and then be sure the data are initialized.

```
// C++ MPI example
#include <mpi.h>

int main( int argc, char * argv[] )
{
    int rank, size;
    int * array = NULL;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    // allocate memory
    array = new int[ ARRAY_SIZE ];
    for( int i = 0; i < ARRAY_SIZE; ++i )
        array[i] = 0;

    // do work
    ...

    // deallocate memory
    delete [] array; array = NULL;

    MPI_Finalize();
}
```

#### 4.4.3. Threaded Codes

As with pure MPI codes, if a thread is to have dynamically allocated memory that only that thread uses, allocate and initialize the memory after the thread has been created to ensure the memory is closest to the core running the thread.

```
// C++ OpenMP example
#include <omp.h>

int main(int argc, char *argv[])
{
    int tid = 0;
    int num_thrds = 1;
    int * array = NULL;

    #pragma omp parallel default(shared) private(tid,num_thrds, array)
    {
        num_thrds = omp_get_num_threads();
        tid = omp_get_thread_num();

        // allocate memory
        array = new int[ ARRAY_SIZE ];
        for( int i = 0; i < ARRAY_SIZE; ++i )
            array[i] = 0;

        // do work
        // ...
    }
}
```

#### 4.4.4. Shared Memory Codes

In shared memory codes, one core may allocate memory that several cores access. If cores that access the same chunk of memory are on the same NUMA, they can all use the same L3 cache and avoid the higher cost of moving data from one NUMA's memory to another NUMA's L3 cache.

## 5. Further Reading

The following documents were used as source material and can provide more in-depth information on the topics covered in this guide.

Cray Online Documentation  
<http://docs.cray.com/>

How to Make Best Use of the AMD Interlagos Processor  
[http://www.hector.ac.uk/cse/reports/interlagos\\_whitepaper.pdf](http://www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf)

AMD Bulldozer Overview, by Ted Barragy  
[http://www.olcf.ornl.gov/wp-content/uploads/2012/01/TitanWorkshop2012\\_Day1\\_AMD.pdf](http://www.olcf.ornl.gov/wp-content/uploads/2012/01/TitanWorkshop2012_Day1_AMD.pdf)

Beagle Cray System Specifications  
<http://wiki.ci.uchicago.edu/Beagle/SystemSpecs>

Blue Waters System Overview  
<https://bluewaters.ncsa.illinois.edu/user-guide>

Maximizing Application Performance on the Cray XT6/XE6, by Jeff Larkin  
<http://www.slideshare.net/jefflarkin/maximizing-application-performance-on-cray-xt6-and-xe6-supercomputers-dodmod-users-group-2010>

Cache Blocking Techniques, by Wendy Doerner  
<http://software.intel.com/en-us/articles/cache-blocking-techniques>

Chapter 6, Optimizing Cache Utilization, from Silicon Graphics  
[http://techpubs.sgi.com/library/dynaweb\\_docs/0640/SGI\\_Developer/books/OrOn2\\_PfTune/sgi\\_html/ch06.html](http://techpubs.sgi.com/library/dynaweb_docs/0640/SGI_Developer/books/OrOn2_PfTune/sgi_html/ch06.html)

Vectorization, presentation by D. Stanzione, L. Koesterke, B. Barth, K. Milfeld  
[http://www.tacc.utexas.edu/c/document\\_library/get\\_file?uuid=7d7b3025-a9fb-41ff-9223-944a8d897149&groupId=13601](http://www.tacc.utexas.edu/c/document_library/get_file?uuid=7d7b3025-a9fb-41ff-9223-944a8d897149&groupId=13601)

Introduction to Intel Advanced Vector Extensions (AVX)  
<http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>

HECToR Guides, Loop Vectorization  
<http://www.hector.ac.uk/cse/documentation/Phase3/>

HECToR Guides, Serial Optimization  
<http://www.hector.ac.uk/cse/documentation/SerialOpt/>