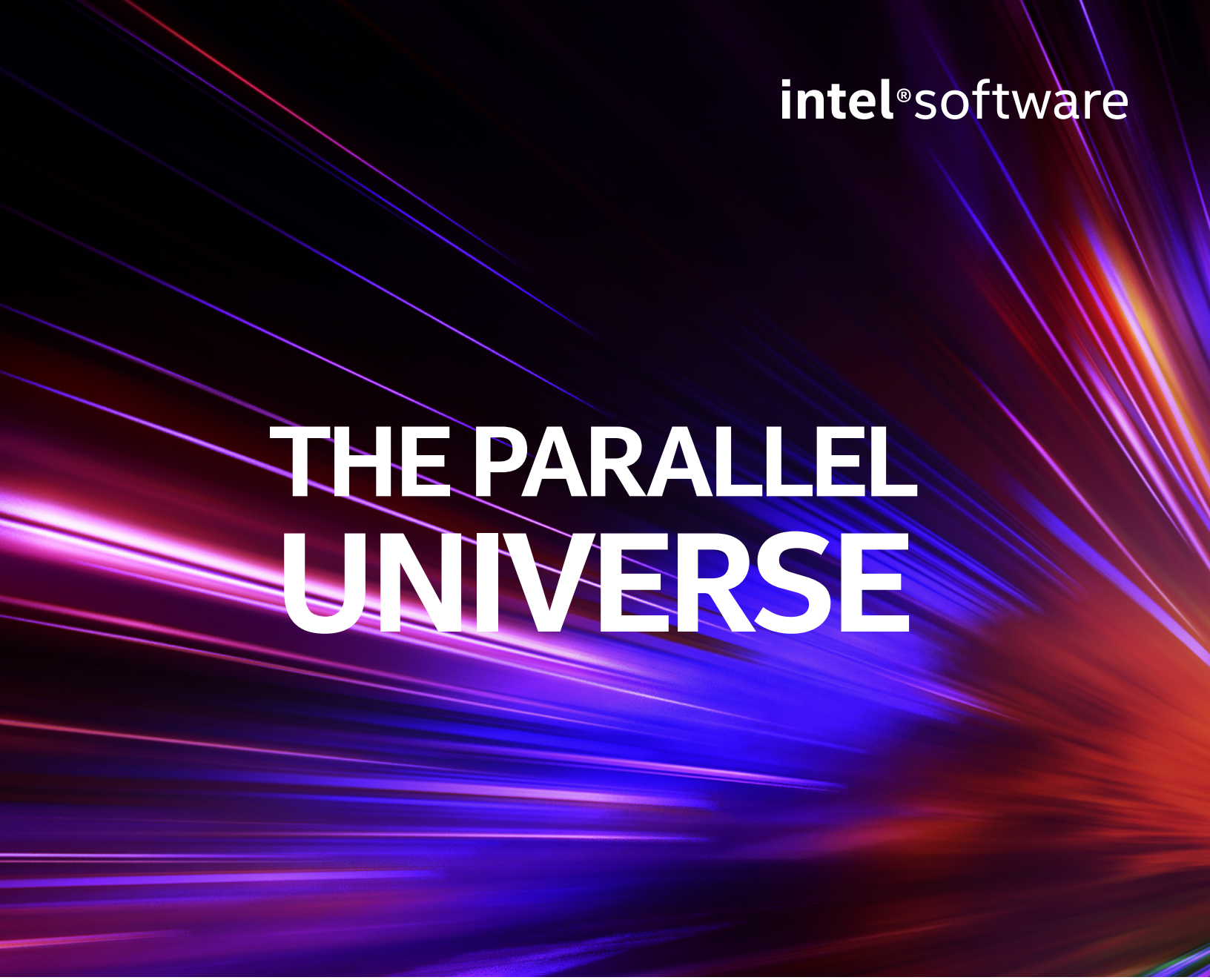


Solving a 2D Heat Equation Using Data Parallel C++

Migrating from CUDA to DPC++ Using the Intel® DPC++ Compatibility Tool

Analyzing Memory and Threading Correctness for GPU-Offloaded Code

Issue
43
2021

Contents

For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

Letter from the Editor

Henry A. Gabb, Senior Principal Engineer at Intel Corporation, is a longtime high-performance and parallel computing practitioner who has published numerous articles on parallel programming. He was editor/coauthor of "Developing Multithreaded Applications: A Platform Consistent Approach" and program manager of the Intel/Microsoft Universal Parallel Computing Research Centers.

Happy New Year...and Goodbye 2020

Welcome to the first issue of 2021. Here's hoping the new year is better..

The **oneAPI initiative** continues to gain traction in industry and academia with the release of the **v1.0 specification**. It closed 2020 with strong showings at SC20 and the oneAPI Developer Summit (content is available **online**). The **oneAPI Application Catalog** went live with 240 applications developed by 200 companies. The catalog showcases oneAPI applications from multiple industries and application categories. The **Intel® Academic Program for oneAPI** also launched to help connect universities and students to the oneAPI initiative. Lastly, the new book, ***Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL***, was published by Apress. It was coauthored by none other than James Reinders, founding editor of *The Parallel Universe* magazine. Speaking of James, I'm pleased to announce that he has rejoined Intel to help promote the oneAPI approach to heterogeneous parallel programming.

Given the increasing popularity of oneAPI, our feature article is a porting case study of a real scientific computing application: **Solving a 2D Heat Equation Using Data Parallel C++**. This is followed by tutorials on **Migrating from CUDA* to DPC++ Using the Intel® DPC++ Compatibility Tool** and **Analyzing Memory and Threading Correctness for GPU-Offloaded Code Using Intel® Inspector**. These three articles will help you get started with DPC++ programming.

From there, we move to code modernization and high-performance computing with our continuing series on taking advantage of compiler optimization reports and tuning MPI applications with **Uncovering More Tuning Opportunities with Intel Compiler Optimization Reports** and **Cluster-Wide MPI Tuning Using Intel MPI Library**, respectively. These articles demonstrate simple ways to boost performance with little or no code modifications.

For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

We close this issue with two articles on improving machine learning performance in data analytics applications. **Accelerating Linear Models for Machine Learning** and **Improving the Performance of XGBoost and LightGBM Inference** describe how to optimize model training and inference using the Intel® Distribution for Python. Both of these articles are republished from **Medium – Intel Analytics Software**, which focuses on Intel software for data science.

As always, don't forget to check out **Tech.Decoded** for more information on Intel solutions for code modernization, visual computing, data center and cloud computing, data science, systems and IoT development, and heterogeneous parallel programming with oneAPI.

Henry A. Gabb

January 2021

For more complete information about compiler optimizations, see our Optimization Notice.

Solving a 2D Heat Equation Using Data Parallel C++ (DPC++)

A Step-by-Step Case Study Porting a C and MPI Application to DPC++

Graham McKenzie, Systems Field Application Engineer, Intel Corporation

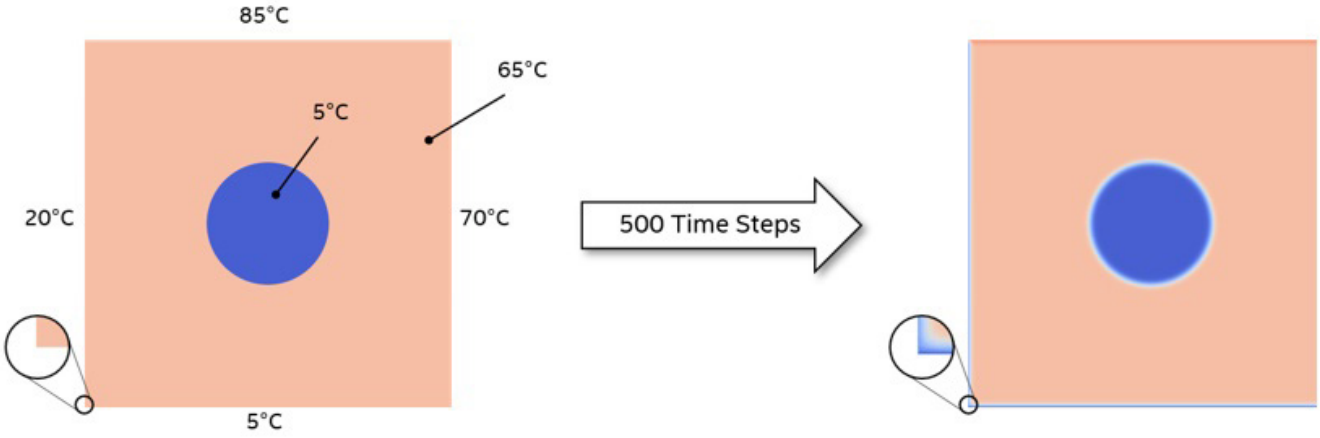
The heat equation is a problem commonly used in parallel computing tutorials. In fact, we start from one such exercise published by the Partnership for Advanced Computing in Europe (PRACE). The original code[1] describes a C and MPI implementation of a 2D heat equation, discretized into a single point stencil (**Figure 1**). The 2D plane is divided into cells, with each cell being updated every timestep based on the previous values of itself and its four neighbours. A more detailed explanation of the problem can be found on the PRACE repository[2].

For more complete information about compiler optimizations, see our Optimization Notice.

$$c_t = c_{t-1} + \alpha \Delta t \left(\frac{l_{t-1} - 2c_{t-1} + r_{t-1}}{\Delta x^2} + \frac{d_{t-1} - 2c_{t-1} + u_{t-1}}{\Delta y^2} \right)$$

1 The 2D heat equation and single point stencil

The default implementation starts with a 2000 x 2000 cell plane that evolves over 500 steps in time (**Figure 2**). The plane is initialized to a uniform temperature except for a disc in the center that has a different uniform temperature. External edges are fixed at four different temperatures.



2 Setting up the computation

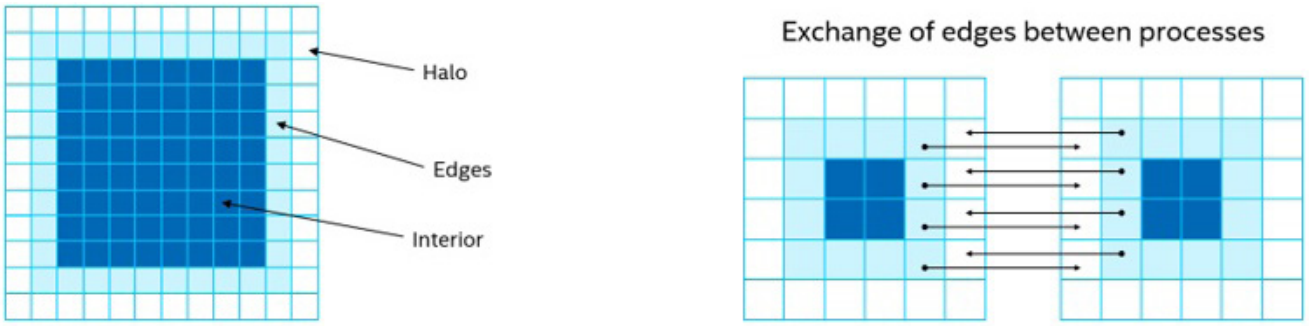
For more complete information about compiler optimizations, see our Optimization Notice.

In this article, readers will learn how we can take code written in standard C/C++, add some simple DPC++ constructs, and run on different parallel processing units. Familiarity with DPC++, SYCL, and oneAPI is assumed.

Initial Port to DPC++

The original PRACE code features MPI constructs to divide the problem into tiles that are processed across multiple processors and multiple nodes. These are initially removed from the code to focus on the core computation. They will be reintroduced later. Other features, such as regular writing of output images and restart checkpoints, are also removed to simplify the problem. The code associated with writing images depicting the initial and final planes were kept as a useful functional verification tool.

A further simplification made initially was to update the whole of the plane in a single kernel. The MPI-based code separates the edges and interior so that the interior can be updated while the halo is being copied from other processes. The edge from one tile becomes the halo from a neighboring tile. Once the halo is updated, the edges can be computed, which are dependent upon both the halo and the interior (**Figure 3**). In this initial port, the edges and interior are treated as one.



3 **Data layout and exchange**

DPC++ Headers and Namespace

In order to use DPC++, two header files need to be included. The first provides support for the DCP++ language, is provided with the DPC++ compiler, and is referenced in `main.c`, `core.c`, and `heat.h`. The second header file is taken from the collection of DPC++ sample programs[3] and is included for the exception handler; this header file is only referenced from `main.c`. We also declare the `sycl` namespace to simplify SYCL constructs in the code body:

```
#include <CL/sycl.hpp>
#include "dpc_common.hpp"

using namespace sycl;
```

For more complete information about compiler optimizations, see our Optimization Notice.

Other changes to `main.c` involve the main time iteration loop, which is wrapped top and tail with DPC++ code. The following five code segments represent contiguous code but are separated here to aid the description.

Device and Queue

A default selector is chosen, which means that the runtime will select the target device to run the kernel on. Usually this will result in a GPU being used, if present, or the host CPU otherwise. If we wanted to force a certain device, we could use `cpu_selector` or `gpu_selector` instead of `default_selector`. A `queue` is then defined based upon the selector and an exception handler which is wrapped in a `try-catch` block.

```
default_selector d_selector;
try {
  queue q(d_selector, dpc::exception_handler);
```

Buffers

Buffers provide containers for data that is present on the target device and are familiar to OpenCL programmers. We take the size of the problem from the data structure found in the original code, adding two to each dimension for the halo (one cell on each side) (**Figure 3**). `global_range` is declared as a one-dimensional range based upon our problem size and dimensionality of the original host array. The buffers are then defined, referencing this range and pointers to the host data:

```
size_t num_items = (current.nx + 2) * (current.ny + 2);
auto global_range = range<1> (num_items);
buffer current_buf(current.data, global_range);
buffer previous_buf(previous.data, global_range);
```

Queue Submission and Accessors

The time evolution for-loop is unmodified from the original code. For each iteration of the loop, we make a queue submission based upon the queue defined above. Accessors define how the buffers can be accessed on the device. In this case, we simply declare `read_write` access for both of our buffers. Although only one buffer is read from and one is written to during each loop iteration, we swap the buffers while they remain in context so we cannot declare them as `read` or `write`.

For more complete information about compiler optimizations, see our Optimization Notice.

```
/* Time evolve */
for (iter = iter0; iter < iter0 + nsteps; iter++) {
  q.submit([&](handler &h) {
  auto current_accessor =
    current_buf.get_access<sycl::access::mode::read_write>(h);
  auto previous_accessor =
    previous_buf.get_access<sycl::access::mode::read_write>(h);
```

Kernel Execution

We use a `parallel_for` kernel, meaning that the body of this section will be submitted for every item in our problem (i.e., every cell in the plane, including the halo in this case). A one-dimensional `id` is defined that will be passed to the kernel identifying the cell being calculated. We could have used a two-dimensional ID, which would aid indexing within the kernel body, but opted for a one-dimensional ID to show a functional port with minimal code changes. The body of the kernel is contained within the function `evolve` in a separate file, `core.c`, which is described later. The original code has one call point to the equivalent `evolve` function and then swaps the pointers within the loop. Here we alternate pointers to our accessors on each loop iteration via separate `parallel_for` calls to make buffer management simpler:

```
   if ((iter - iter0) % 2 == 0)
      h.parallel_for({num_items}, [=](id<1> i) {
        evolve(i, current_accessor.get_pointer(),
previous_accessor.get_pointer(),
          current.nx_full, current.ny_full, dx2, dy2, a, dt);
      });
    else
      h.parallel_for({num_items}, [=](id<1> i) {
        evolve(i, previous_accessor.get_pointer(),
current_accessor.get_pointer(),
          current.nx_full, current.ny_full, dx2, dy2, a, dt);
      });
  }); // submit
} // for loop
```

For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

Error Handling

Any errors that occur during kernel execution are passed to the host application scope and are handled there with conventional C++ exception handling techniques. Here, we simply rely upon the standard exception handler that is provided in the sample programs:

```
} catch (exception const &e) {
  std::cout << "An exception is caught.\n";
  std::terminate();
}
```

Kernel Code

The following code is the complete listing for `core.c`. It is vastly simplified from the original code since we are not worried about the inter-process communication or separating the interior from the edge calculation. In the original code, as with traditional CPU programming, the evolve function is called once for the entire plane and two nested for loops traverse the x and y dimensions, calculating the update for each cell. Due to the use of our `parallel_for` call above, the function below will be called for each individual cell. Therefore, there are no for-loops.

The global range used when calling `parallel_for` included the halo, which is not updated but is needed to calculate the new values for the edges. The if statement is used to omit the halo so that we do not attempt to read memory outside of the buffers, which would result in a segmentation fault. The actual calculation is confined to the last line of code and is very similar to the original code. Various intermediate variables are declared purely to aid readability.

There is a subtle change to the function parameters. The most obvious is the inclusion of the ID, which is needed to identify which cell is being calculated. We also pass in `dx2` and `dy2` to the kernel rather than `dx` and `dy`. `dx2` and `dy2` were originally calculated with the kernel, but their values remain constant across all cells. Since our new `evolve` function is called for every cell, calculating these values each time would be an unnecessary overhead. We also need to declare the function as `SYCL_EXTERNAL` to tell the compiler that this is kernel code, something that is not obvious from the body. The function prototype in the header file is also modified accordingly:

For more complete information about compiler optimizations, see our Optimization Notice.

```
#include <CL/sycl.hpp>
#include "heat.h"

SYCL_EXTERNAL void evolve(sycl::id<1> i, double *curr, double *prev, int nx,
  int ny, double dx2, double dy2, double a, double dt) {

  size_t width = ny + 2;
  size_t num_items = (nx + 2) * (ny + 2);

  if (i > width && i % width != 0 && (i+1) % width != 0 && i < num_items -
width) {

      size_t iu = i - width;
      size_t id = i + width;
      size_t ir = i + 1;
      size_t il = i - 1;

      curr[i] = prev[i] + a * dt * ((prev[iu] - 2.0 * prev[i] + prev[id]) / dx2
+
                (prev[ir] - 2.0 * prev[i] + prev[il]) / dy2);
  }
}
```

Compiling the Application and Kernel

The original code uses the `mpicc` compiler wrapper. To compile for DPC++, we use the `dpcpp` compiler and modify the `Makefile` accordingly. However, the files associated with writing out PNG files require a C compiler so we use GCC for this and wrap the function prototype with `extern "C" {…}`. These files reference libpng[4], which can be installed with `sudo apt-get install libpng-dev` or downloaded and built from source.

Targeting FPGA

The code and modified `Makefile` above are appropriate for targeting CPU and GPU. For FPGA we make a few minor changes in the source code and the `Makefile`.

`main.c` is modified to select the `fpga_selector` when FPGA is defined:

```
#include <CL/sycl/INTEL/fpga_extensions.hpp>
…
#if FPGA
INTEL::fpga_selector d_selector;
#else
default_selector d_selector;
#endif
```

For the `Makefile`, there are just a few extra flags for the compiler and linker. We have also changed the executable name so that it remains separate from the CPU/GPU version.

```
CXXFLAGS=-O3 -g -Wall -std=c++17 -fintelfpga -DFPGA
LDFLAGS=-L$(LIBPNG_ROOT)/lib -fsycl-link -Xshardware -fintelfpga
EXE=heat.fpga
```

Running the Application on the Intel® DevCloud

The Intel® DevCloud[5] gives users the opportunity to try oneAPI and DPC++ for free with a preinstalled environment and access to multiple Intel CPU, GPU, and FPGA technologies. The following assumes that access to Intel DevCloud is already available and configured as per the Getting Started Guide[6].

Before we can compile and run our code, we need to install `libpng`. As sudo access is not available on the Intel DevCloud, we do this by compiling from source. Download the source code from Reference 7 and copy to the Intel DevCloud, then perform the following steps:

```
tar xf libpng-1.6.37.tar.gz
cd libpng-1.6.37/
./configure --prefix=/home/<username>/lib/libpng-1.6.37
make check
make install
```

Then add the following to `~/.bashrc`, or run these commands upon any new session. The environment variable `LIBPNG_ROOT` can then be used to reference the library in the `Makefile`:

```
export LIBPNG_ROOT=/home/<user>/lib/libpng-1.6.37
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$LIBPNG_ROOT/lib"
```

To compile and run our heat equation, we can use interactive sessions on the relevant compute nodes for CPU and GPU using one of the following commands, respectively:

```
qsub -I -l nodes=1:skl:ppn=2 -d .
qsub -I -l nodes=1:gen9:ppn=2 -d .
```

For more complete information about compiler optimizations, see our Optimization Notice.

Then, we simply make and run the code before exiting the session:

```
make
./heat
exit
```

Because FPGA can take a long time to compile, it's best to submit this as a batch job, targeting one of the FPGA compile nodes as follows:

```
qsub -l nodes=1:fpga_compile:ppn=2 -d . ./compile_fpga.sh
```

It's also advisable to `make clean` between CPU/GPU compiles and FPGA or compile in a separate directory to avoid conflicts. The contents of `compile_fpga.sh` are as follows and the file must be executable:

```
#!/bin/bash
make -f Makefile.fpga
```

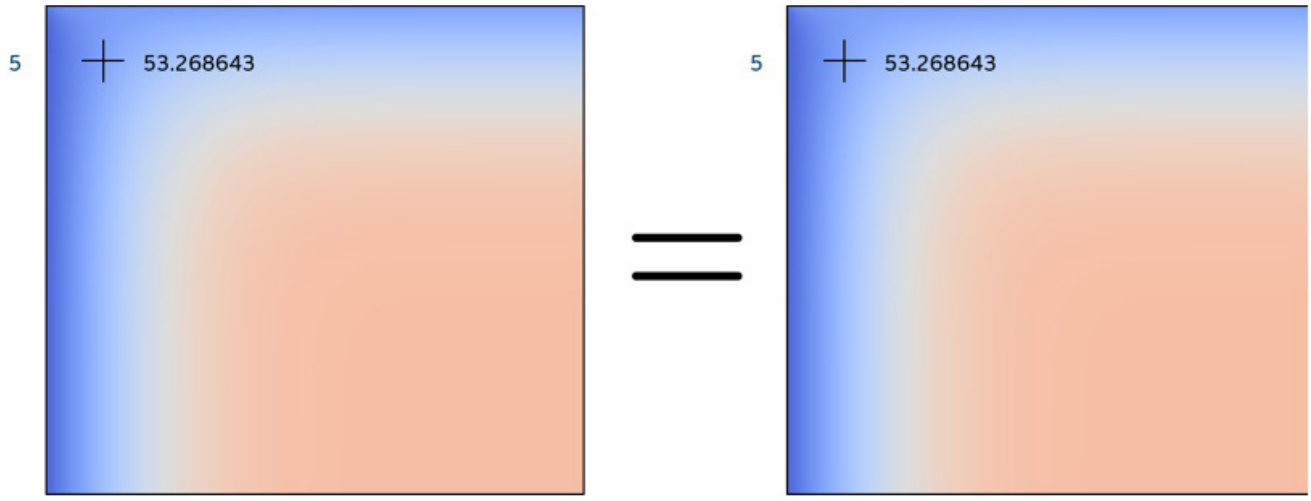
Note that we have kept a separate `Makefile` for the FPGA compilation. The status of the job can be checked using the `qstat` command. Once completed and successful, the kernel can be run on the FPGA using an interactive session, which is launched as follows:

```
qsub -I -l nodes=1:fpga_runtime:ppn=2 -d .
```

Functional Verification

At the end of execution, the host application prints out the final value of a specific cell to the terminal as well as outputting a PNG image of the final plane (**Figure 4**). The specific value provides a quick check against different code iterations and kernel targets. This value is the same across runs on CPU, GPU, and FPGA and matches the value of the original code. The PNG image can be compared using third-party applications and shows no differences across the three technologies or the original code.

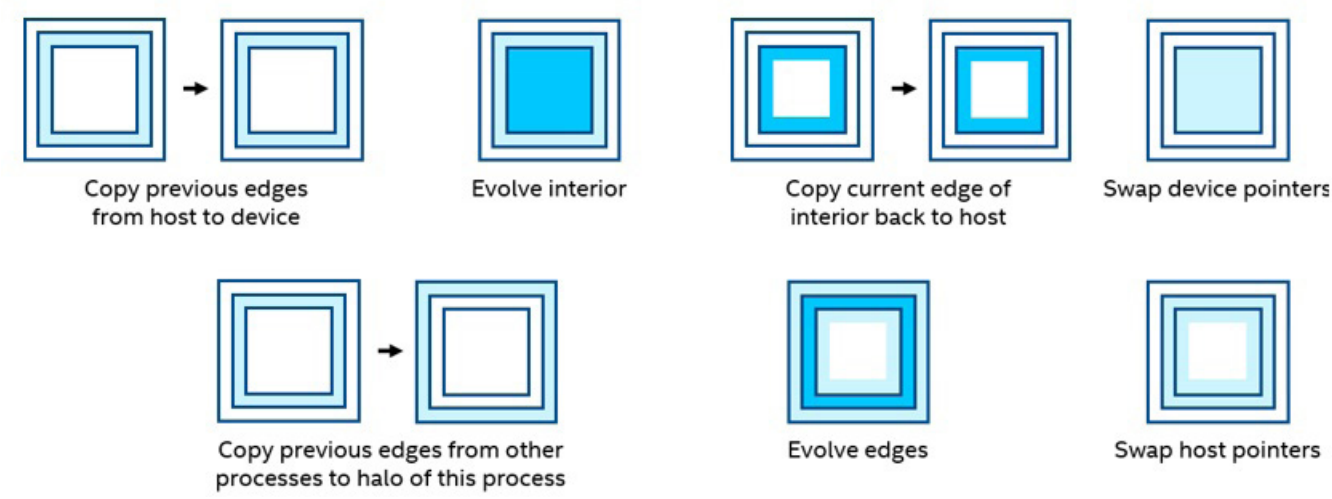
For more complete information about compiler optimizations, see our Optimization Notice.



4 **Single point verification**

MPI Implementation

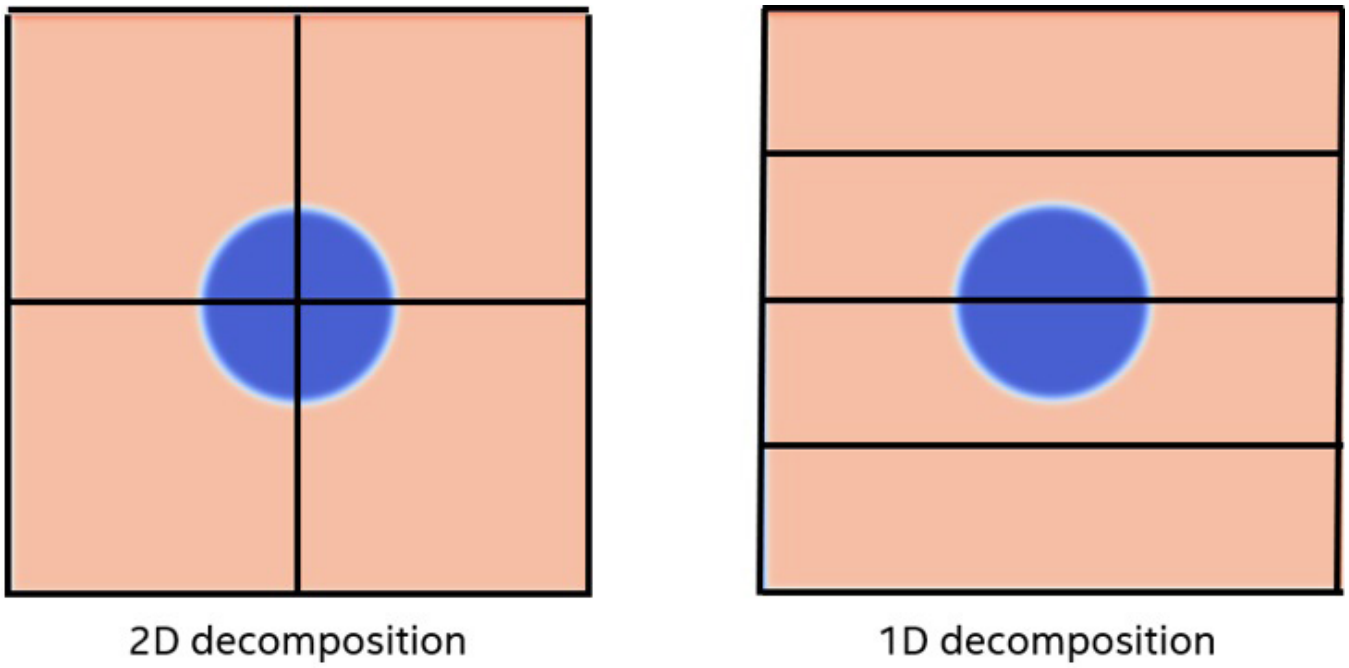
Before adding the MPI multi-process support back into the code, we need to think about how to divide the problem between host and device, as well as how to parallelize the operations. A reasonable starting point is to have the host deal with the edge calculations and transfer between processes and have the device calculate the interior. This also aligns with the approach in the original code. During the time iteration loop, the interior can be evolved independently of the edges with a synchronization point prior to the swapping of pointers at the end of each iteration (**Figure 5**). The two rows can operate in parallel.



5 **Data exchange between host and device during each iteration**

For more complete information about compiler optimizations, see our Optimization Notice.

Another important consideration is how the problem is decomposed among MPI processes. The original code uses a two-dimensional decomposition, meaning that with four processes, the problem will be divided into four quadrants (2 x 2 tiles). The data transfers between host and device will be complex and comprise many individual PCIe transactions for the left and right edges, if we transfer only the data needed. It would be more efficient to transfer the whole data planes back and forth between each iteration of the interior evolution on the device. This is because one large transfer is faster than many small transfers. However, a better way would be to decompose the problem in one dimension only. This way, we can transfer only the data needed between host and device and those transfers will require just two lots of contiguous memory (top and bottom edges) (**Figure 6**). **Figure 7** shows the relative execution time normalized to the time taken to transfer all data. Note that because the global edges of this problem are fixed, there is no need to transfer left and right edges for the one-dimensional implementation, so the amount of data transferred for 1D and 2D decompositions is the same.



6 **1D versus 2D data decomposition**

Data Parallel C++
A Standards-Based, Cross-Architecture Language
Get Started

For more complete information about compiler optimizations, see our Optimization Notice.

7 **Relative execution time**

We can force a one-dimensional decomposition by simply changing the dimensions passed to the `MPI_Dims_create` function in `setup.c`. A zero passed to this function allows the runtime to define the range of each dimension. By using a one as the second value, we force a single column implementation:

```
//int dims[2] = {0, 0};
int dims[2] = {0, 1};
```

The buffer implementation used in our previous implementation transfers data to the device implicitly prior to execution and then back again when the buffers go out of context. To be able to transfer data back and forth within the loop, we need finer control. The Unified Shared Memory (USM) model allows data to be allocated as device, host, or shared and enables familiar C++ constructs to interact with memory. We will use device allocated memory, which enables explicit control over data transfers between host and device.

Taking the original code as the starting point and keeping all the MPI code, the following changes are made. The header and namespace changes are omitted here, since they are the same as before. The following five sections represent contiguous code.

For more complete information about compiler optimizations, see our Optimization Notice.

Device Memory Allocation

The device selector and queue are as per the first implementation. Instead of using buffers we use `malloc_device` to explicitly define memory on the device, which is then copied over with explicit `memcpy` functions. The queue needs to be referenced to provide the device context. The `wait` function waits for the previous memory operations to complete. This is necessary because host execution continues after submitting the requested action to allow parallel operation between host and accelerator. The `wait` function serves as a synchronization point to ensure that all actions in the queue, which may execute out of order, are complete before host execution continues. In the previous example, the SYCL runtime took care of scheduling operations using buffers.

```
default_selector d_selector;
try {
  queue q(d_selector, dpc::exception_handler);

  size_t num_items = (current.nx + 2) * (current.ny + 2);
  auto dev_curr = malloc_device<double>((num_items), q);
  auto dev_prev = malloc_device<double>((num_items), q);

  q.memcpy (dev_curr, current.data, num_items * sizeof(double));
  q.memcpy (dev_prev, previous.data, num_items * sizeof(double));
  q.wait();
```

Copy Edges to Device

At the beginning of each timestep, the top and bottom edges of each slice are copied from host to device using pointer math to identify the edges, which are one row below and above the top and bottom, respectively, due to the halo. Note that the copy operation is initiated, then the exchange function, and then the wait so that the copy and exchange can operate in parallel.

For more complete information about compiler optimizations, see our Optimization Notice.

```
/* Time evolve */
 for (iter = iter0; iter < iter0 + nsteps; iter++) {

   q.memcpy (dev_prev + previous.ny+2, previous.data+previous.ny+2,
             (previous.ny+2) * sizeof(double));
   q.memcpy (dev_prev + (previous.ny+2) * previous.nx, previous.data +
             (previous.ny+2) * previous.nx, (previous.ny+2) * sizeof(double));

   exchange_init(&previous, &parallelization);
   q.wait();
```

Kernel Execution

In this implementation, the device evolves the interior rather than the whole plane and so the function name for the compute is changed accordingly. The other subtle difference is that the device memory pointers are passed in instead of the pointers to buffer accessors. The `exchange_finalize` function is from the original code, which waits for the MPI-based data exchange to complete before evolving the edges. We then wait for the kernel to complete before copying the edge of the interior back to the host.

```
  q.submit([&](sycl::handler &h) {
      h.parallel_for(num_items, [=](id<1> i) {
        evolve_interior(i, dev_curr, dev_prev, current.nx, current.ny, dx2,
dy2, a,
                        dt);
      });
    });

    exchange_finalize(&parallelization);
    evolve_edges(&current, &previous, a, dt);

    q.wait();
    q.memcpy (current.data + (current.ny+2) * 2, dev_curr + 2 * (current.ny+2),
              (current.ny+2) * sizeof (double));
    q.memcpy (current.data + (current.ny+2) * (current.nx - 1) , dev_curr +
              (current.ny+2) * (current.nx -1), (current.ny+2) * sizeof
(double));
    q.wait();
```

For more complete information about compiler optimizations, see our Optimization Notice.

Pointer Swap

IThe final step for each time iteration is to swap the pointers. On the host this is handled by the `swap_fields` function from the original code. For the device, we perform a simple pointer swap.

```
    swap_fields(&current, &previous);
     double *dev_tmp = dev_curr;
     dev_curr = dev_prev;
     dev_prev = dev_tmp;
   } // for loop
```

Completion

To finalize the kernel operation, we copy both planes back to the host, free the previously allocated device memory and catch any exceptions that may have occurred.

```
device memory and catch any exceptions that may have occurred.
q.memcpy (current.data + 2 * (current.ny+2), dev_curr + 2 * (current.ny+2),
          (current.ny+2) * (current.nx-2) * sizeof (double));
q.memcpy (previous.data + 2 * (previous.ny+2), dev_prev + 2 * (previous.ny+2),
          (previous.ny+2) * (previous.nx-2) * sizeof (double));
q.wait();

free(dev_curr, q);
free(dev_prev, q);

} catch (exception const &e) {
  std::cout << "An exception is caught." << std::endl;
  std::terminate();
}
```

Kernel Code

Kernel code looks very similar to the previous implementation. The only difference is in the `if` statement, which now excludes both the halo and edges from the top and bottom, but only the halo from the left and right edges.

For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

```
SYCL_EXTERNAL void evolve_interior(sycl::id<1> i, double *curr, double *prev,
  int nx, int ny, double dx2, double dy2, double a, double dt)
{
  size_t width = ny + 2;
  size_t num_items = (ny + 2) * (nx + 2);

  if (i > (width*2) && (i % width) > 0 && (i % width) < (width-1) && i <
     (num_items - width*2)) {

    int iu = i - width;
    int id = i + width;
    int ir = i + 1;
    int il = i - 1;

    curr[i] = prev[i] + a * dt * ((prev[iu] - 2.0f * prev[i] + prev[id]) / dx2
+
                                  (prev[ir] - 2.0f * prev[i] + prev[il]) /
dy2);
  }
}
```

Additional Changes

Since the left and right edges are now evolved using the device kernel, we need to remove this operation from the host CPU in the `evolve_edges` function. This function comprises four for-loops, one for each edge. The left and right edges are the last two, which we simply comment out.

Compiling and Running with MPI

We use the Intel dpcpp compiler but add additional flags to include and link in the Intel MPI library:

```
CXXFLAGS=-O3 -Wall -std=c++17 -I$(I_MPI_ROOT)/intel64/inc
LDFLAGS=-L$(LIBPNG_ROOT)/lib -L$(I_MPI_ROOT)/intel64/lib
LIBS=-lpng -lm -lmpi
```

To run the code, we use `mpirun` and pass in the number of processes and the executable:

```
mpirun -np 4 ./heat
```

Both the single-point value and PNG comparisons confirm that this implementation functionally matches the previous implementation and the original code. This remains true when run on a single node with multiple processes and across multiple nodes using multiple GPUs.

For more complete information about compiler optimizations, see our Optimization Notice.

Conclusions

With a few basic changes to the code, we have managed to take a standard problem and convert it to a DPC++ implementation using buffers and accessors. We can use the Intel DevCloud to compile code and run on multiple targets including CPU, GPU, and FPGA, with each of those targets producing the same functional results as the original implementation. Finally, we showed how to use USM to provide explicit control of memory transfers between the host and device, necessary to manage the data flow between multiple processes using MPI.

References

1. **PRACE MPI Implementation of Two-Dimensional Heat Equation using MPI and C**
2. **PRACE Description of the Two=Dimensional Heat Equation**
3. **oneAPI Direct Programming Samples**
4. **LibPNG**
5. **Intel DevCloud**
6. **Intel DevCloud Start Guide**
7. **LibPNG Source Code**

NEWS HIGHLIGHTS

Intel Releases oneAPI Toolkits for XPU Software Development

Intel released version 2021.1 of its oneAPI Toolkits to simplify development of high-performance applications across Intel® CPUs, GPUs, and FPGAs. The toolkits combine Intel's rich heritage of proven developer tools with oneAPI, an open, standards-based, unified cross-architecture programming model, to enable developers to break free of the economic and technical burdens of proprietary programming models with the freedom to choose the best hardware for their specific workloads.

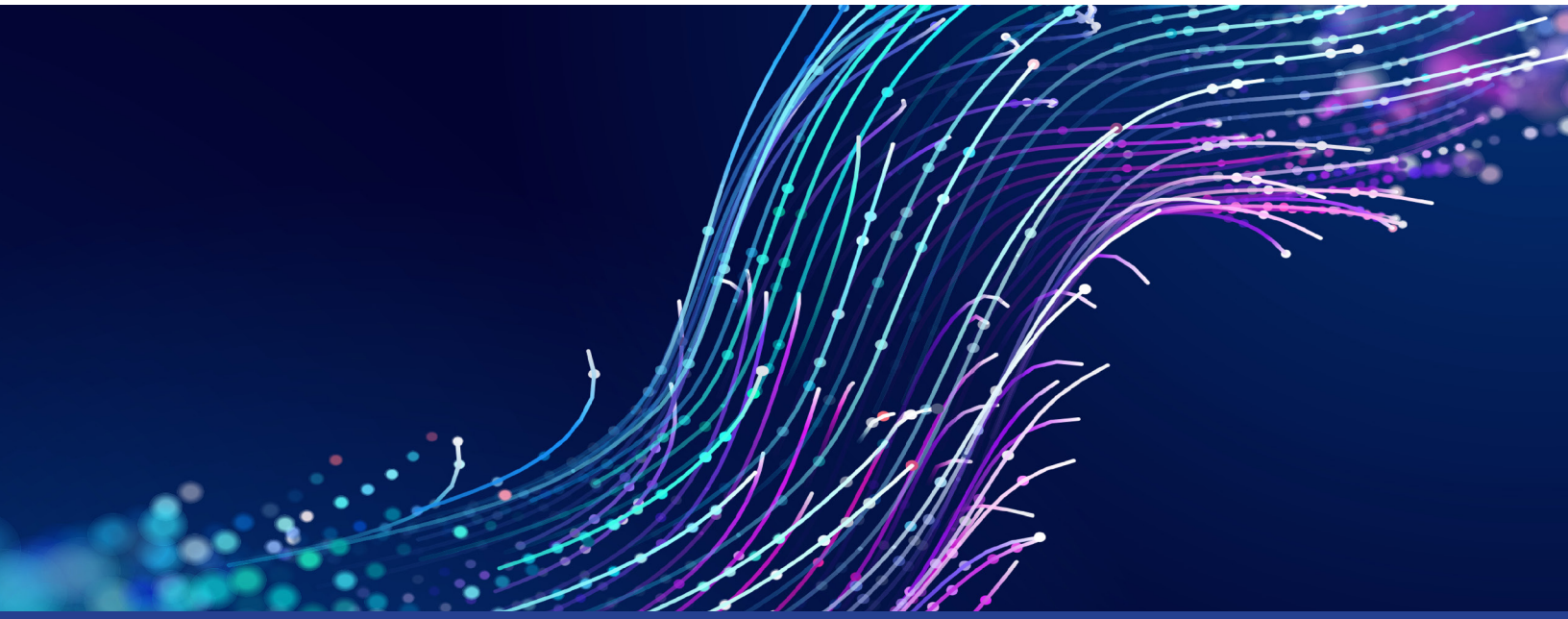
Read on >

For more complete information about compiler optimizations, see our Optimization Notice.

Deliver uncompromised performance for diverse workloads across multiple architectures with oneAPI.

Learn How >

For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-not
Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
© Intel Corporation



Migrating from CUDA to DPC++ Using the Intel® DPC++ Compatibility Tool

It's Easier than Ever to Move to a Non-Proprietary Programming Language

Subarnarekha Ghosal, Compiler Technical Consulting Engineer, Intel Corporation

It's becoming apparent that future computing systems will be heterogeneous. The **MAGMA** project at the University of Tennessee is developing a dense linear algebra library similar to LAPACK but for heterogeneous architectures, like current CPU and GPU systems. I was looking for a sparse solver code sample that gives good performance across different architectures. MAGMA was an obvious candidate. This article describes my use of the Intel® DPC++ Compatibility Tool (DPCT) to migrate MAGMA CUDA code to Data Parallel C++ (DPC++).

For more complete information about compiler optimizations, see our Optimization Notice.

Migration Steps and Hacks Required

Migration can happen in two ways. The first method is file-to-file manual migration, which is a good choice if you're just migrating a few files. The second method is to create a .json file for projects that use make or cmake. MAGMA has a makefile, so let's focus on the JSON approach.

The build options of input projects files (e.g., include path, macros definitions, etc.) are collected in a .json file. It is mainly generated using the `intercept-build make` command. (Be sure you're using Make 4.0 or later.) Next, we run the `dpct` command, which has a handful of flags to help with migration. The command-line for migrating MAGMA was:

```
dpct -p=compile_commands.json --out-root=<dpct_output_path> --in-
root=<path_to_application>
--keep-original-code --process-all
```

Table 1. Flags and functions

Flags	Functions
`-p=compile_commands.json`	Specifies the `compile_command.json` file.
`--keep-original-code`	To see the lines of code that has been changed by the compatibility tool, it's a good practice to use this flag. This flag allows you to see the original code followed by the code that has changed in the migrated file by the Intel® DPC++ Compatibility Tool.
`--in-root=<path to input folder>`	Specifies the directory path for the root of the source tree to be migrated.
`--out-root=<path to dpct_output folder>`	Used to specify a custom path to the output directories.
`--process-all`	The Intel DPC++ Compatibility tool skips file(s) if the file(s) doesn't contain any syntax or type that needs to be migrated, and the skipped file(s) are not available in the out-root folder. This option ideally replicates the folder structure in out-root that exists in the original application.
`--cuda-include-path=<dir>`	When CUDA headers are not in the expected path or there are multiple versions of CUDA in the same path, it is a good practice to use this flag to specify the path to the desired version of CUDA.

For more complete information about compiler optimizations, see our Optimization Notice.

(See **the documentation** for additional flags.)

The next step is to interpret the output after running `dpct` on the application. The `dpct` annotates places in the code where modifications may be necessary to make the code DPC++ compliant or syntactically correct, e.g:

```
/path/to/file.hpp:26:1: warning: DPCT1003:0: Migrated API does not return
error code. (*,0) is inserted.
You may need to rewrite this code.
// source code line for which the warning was generated
^
```

For large projects, it's advisable to redirect the migration logs to a file. Learn how various error codes/ diagnostics are reported by the tool **here**.

Figures 1 and **2** show a successful migration of a kernel call from the MAGMA library. Since the migration is done with the `--keep-original-code` flag, the original code is also present in the migrated file (**Figure 2**).

```
{
    dim3 grid( magma_ceildiv( m, BLOCK_SIZE ) );
    magma_int_t threads = BLOCK_SIZE;
    cgeellmv_kernel_shift<<< grid, threads, 0, queue->cuda_stream() >>>
                ( m, n, nnz_per_row, alpha, lambda, dval, dcolind, dx,
                                    beta, offset, blocksize, addrows, dy );


    return MAGMA_SUCCESS;
}
```

1 **Original CUDA code**

```
{
/* DPCT_ORIG     dim3 grid( magma_ceildiv( m, BLOCK_SIZE ) );*/
    sycl::range<3> grid(1, 1, magma_ceildiv(m, BLOCK_SIZE));
    magma_int_t threads = BLOCK_SIZE;
/* DPCT_ORIG     cgeellmv_kernel_shift<<< grid, threads, 0, queue->cuda_stream()
   >>> ( m, n, nnz_per_row, alpha, lambda, dval, dcolind, dx, beta, offset,
   blocksize, addrows, dy );*/
    /*
    DPCT1049:2224: The workgroup size passed to the SYCL kernel may exceed the
    limit. To get the device limit, query info::device::max_work_group_size.
    Adjust the workgroup size if needed.
    */
 queue->cuda_stream()->submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::nd_range<3>(grid * sycl::range<3>(1, 1, threads),
                                     sycl::range<3>(1, 1, threads)),
                   [=](sycl::nd_item<3> item_ct1) {
                    cgeellmv_kernel_shift(m, n, nnz_per_row, alpha, lambda,
                                          dval, dcolind, dx, beta, offset,
                                          blocksize, addrows, dy, item_ct1);
                   });
 });

    return MAGMA_SUCCESS;
}
```

2 Migrated DPC++ code

Some manual effort is required to migrate functions that the tool can't migrate, but annotations generated by the tool make this easier:

```
/*
DPCT1050:3634: The template argument of the image_accessor_ext could
not be deduced. You need to update this code.
*/
```

There are sometimes multiple annotations for a line of CUDA code:

```
/* DPCT_ORIG             zmgesellptmv_kernel_1_3D_tex<true><<< grid, block, 0,
   queue->cuda_stream() >>> ( m, n, num_vecs, blocksize, alignment, alpha,
              dval, dcolind, drowptr, texdx, beta, dy );*/
           /*
           DPCT1049:1518: The workgroup size passed to the SYCL kernel may
           exceed the limit. To get the device limit, query
           info::device::max_work_group_size. Adjust the workgroup size if
           needed.
           */
           /*
           DPCT1050:3634: The template argument of the image_accessor_ext could
           not be deduced. You need to update this code.
           */
```

For more complete information about compiler optimizations, see our Optimization Notice.

Some CUDA libraries have equivalent functions in oneAPI libraries (e.g., oneMKL, oneDNN, oneVPL, etc.). The Intel DPC++ Compatibility Tool can migrate many CUDA library functions to their oneAPI equivalents. The tools will call out those that can't be migrated directly. It's often possible to manually implement the same functionality. For example, the CUDA `cusparseDcsrmv` function can be manually migrated using a combination of oneMKL `mkl::sparse::gemv` and `mkl::sparse::set_csr_data` functions.

The Intel DPC++ Compatibility Tool is evolving every day with input from users. Some of the known issues with the tool are stated in the Known Issues and Limitations Section of the **release notes**.

The Intel DPC++ Compatibility Tool reduces the time and effort required to migrate CUDA applications to DPC++. It gives helpful annotations and warnings to minimize the manual effort required for sections of code that are not migrated by the tool. Migrating MAGMA from CUDA to DPC++ would have been a tedious job without this tool.

If you're interested in migrating a CUDA application to DPC++, **online training** can help you get started. The Intel DPC++ Compatibility Tool is available in the **Intel® oneAPI Base Toolkit**

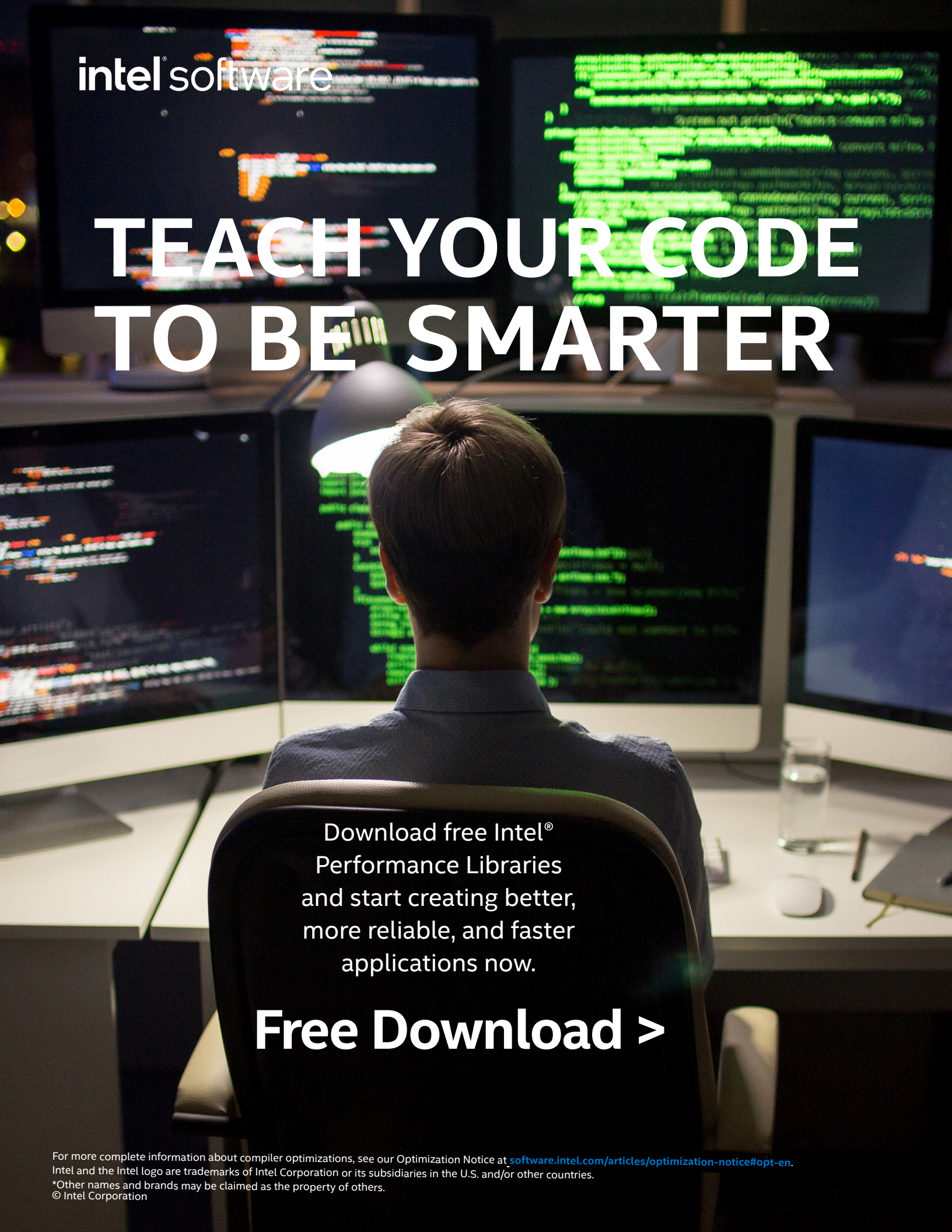
NEWS HIGHLIGHTS

Lobachevsky State University of Nizhni Novgorod to Accelerate Studies of Quantum Processes Using oneAPI

Lobachesky State University of Nizhni Novgorod (UNN) announced a new oneAPI Center of Excellence (CoE) to facilitate studies in contemporary physics using the power of CPUs, GPUs, and other accelerators with oneAPI cross-architecture programming. This new center aims to address research challenges requiring high-performance computing (HPC) on heterogeneous architectures,

Read on >

For more complete information about compiler optimizations, see our Optimization Notice.



For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
© Intel Corporation

Analyzing Memory and Threading Correctness for GPU-Offloaded Code

Intel® Inspector Makes It Easy to Debug Heterogeneous Parallel Code

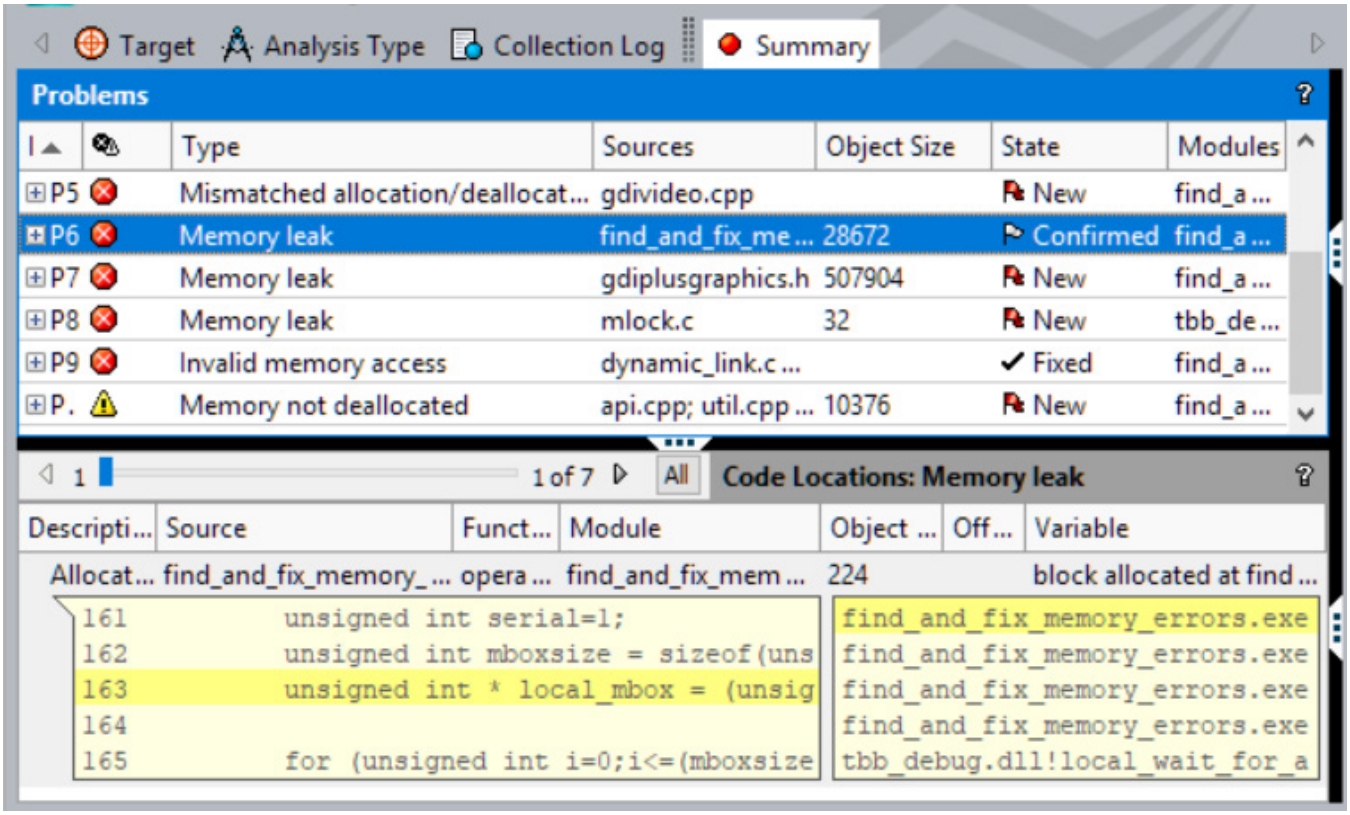
Kevin O'Leary, Lead Technical Consulting Engineer, and Michael Tutin, Software Architect, Intel Corporation

Modern workloads are diverse—and so are architectures. No single architecture is best for every workload. Maximizing performance takes a mix of scalar, vector, matrix, and spatial architectures deployed in CPU, GPU, FPGA, and other future accelerators. Heterogeneity adds complexity that can be difficult to debug. This article introduces the new features of **Intel® Inspector** that support the analysis of code that's offloaded to accelerators.

For more complete information about compiler optimizations, see our Optimization Notice.

Intel® Inspector Overview

Memory errors and nondeterministic threading errors are difficult to find without the right tool. Intel Inspector is designed to find these errors. It's a dynamic memory and threading error debugger for C, C++, DPC++, and Fortran applications that run on Windows or Linux operating systems.



1 **Intel® Inspector in action**

Figure 1 shows the types of problems that Intel Inspector finds:

- **Memory errors** including leaks, invalid access, and more
- **Persistent memory errors** such as missing or redundant cache flushes
- **Threading errors** such as data races and deadlocks

It's easy to use, reliable, and accurate. No special recompilation is required. You can use your normal debug or production build to catch and debug the errors. Intel Inspector can analyze dynamically generated or linked code and inspect third-party libraries, even when source code isn't available. It breaks into the debugger just before the error occurs. Automated regression analysis is possible using the command-line option.

For more complete information about compiler optimizations, see our Optimization Notice.

How to Analyze Your Offloaded Code Using Intel Inspector

Correctness analysis is more complicated when offloading code to an accelerator. Our experiences with DPC++ uncovered the need for a tool to assist in debugging offload issues. The current version of Intel Inspector introduces an important approach called "early interception." It means that for offloaded code, it intercepts some problems in the early stage before kernel execution. **Tables 1** and **2** list the offload issues that can be detected using Intel Inspector. Data races on shared data are reported, but there are limitations:

- **DPC++ barriers and OpenMP synchronizations are ignored.** The tool will report false positives even if work-items are synchronized.
- **Data races are not detected** on variables defined in kernel local memory.
- **The instructions below set up your application to run on your CPU**, but some GPU analysis is supported using early interception.

Step 1 is to configure your application to run on the host CPU:

```
export SYCL_BE=PI_OPENCL
export SYCL_DEVICE_TYPE=CPU
```

Next, configure OpenMP applications to run kernels on the CPU device:

```
export OMP_TARGET_OFFLOAD=MANDATORY
export LIBOMPTARGET_DEVICETYPE=cpu
```

Verify that the application works correctly before running the analysis. Enable code analysis and tracing in the JIT compiler/runtimes:

```
export CL_CONFIG_USE_VTUNE=True
export CL_CONFIG_USE_VECTORIZER=false
```

Set up the Inspector environment

```
source /opt/inspxe/inspxe-vars.sh
```

Step 2 is to run an analysis on a small workload using either the GUI (`inspxe-gui`) or command-line (`inspxe-cl`) clients. Perform analysis using the command-line client as follows:

For more complete information about compiler optimizations, see our Optimization Notice.

```
inspxe-cl -c mi3 -- <app> [app_args]
```

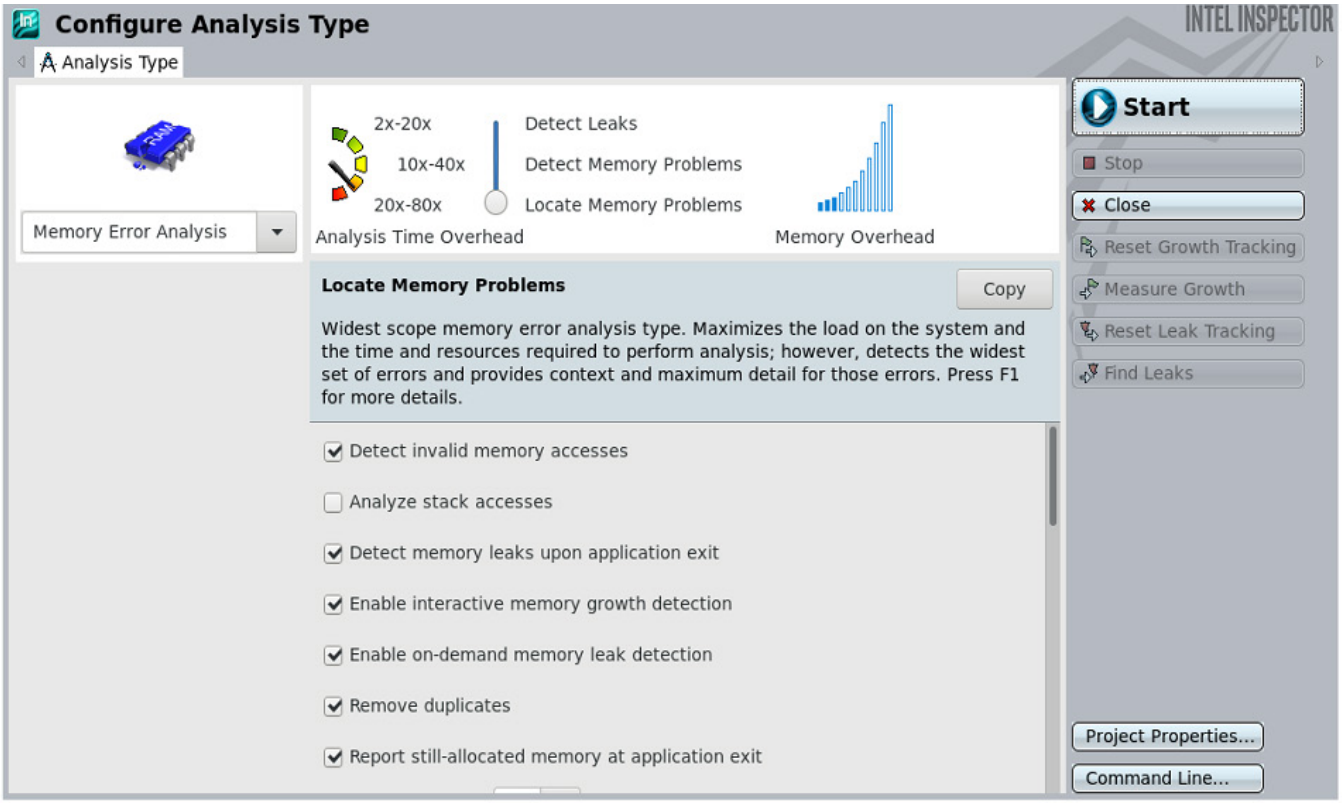
View the results as follows:

```
inspxe-cl -report=problems -report-all
```

Alternatively, you can view the results in the GUI:

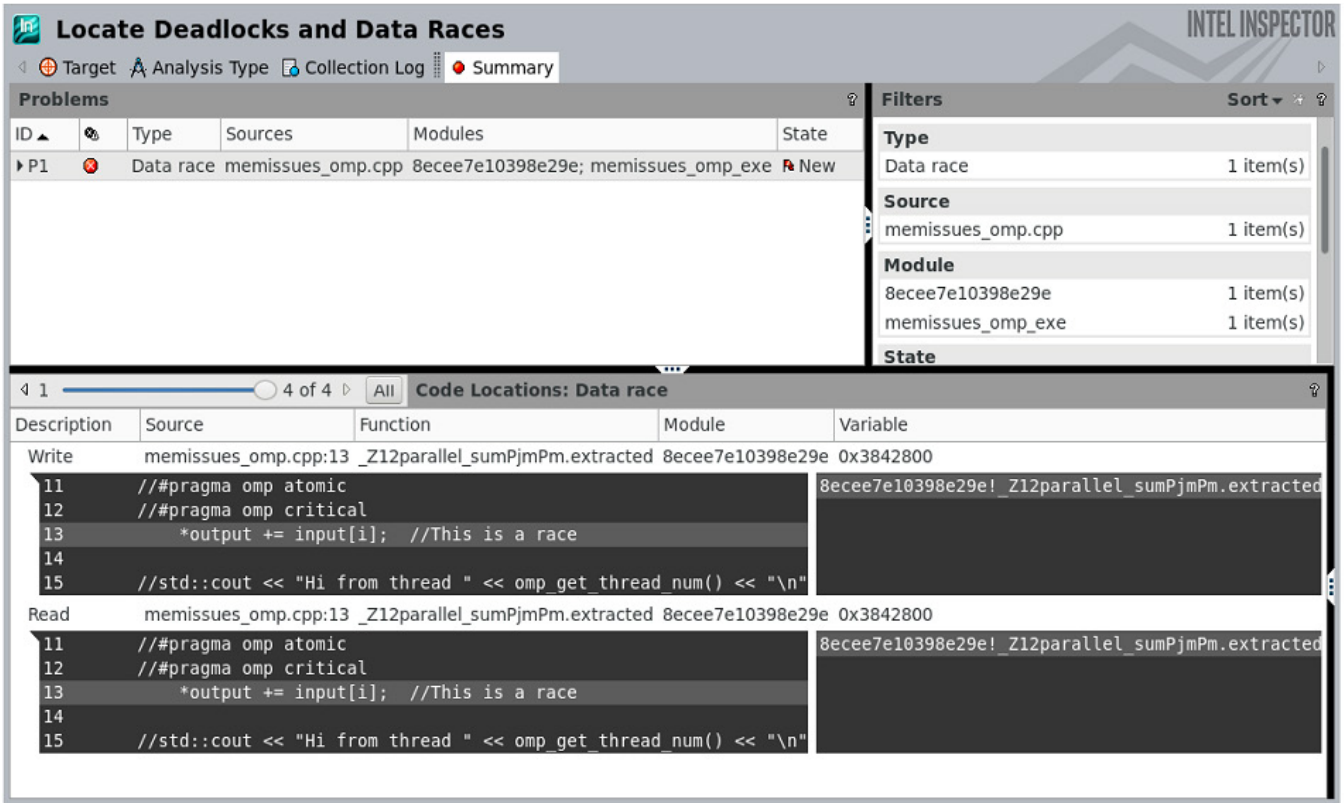
```
inspxe-gui <result folder>
```

You can also launch an analysis and view the results in the GUI (**Figures 2** and **3**).



2 **To launch an analysis in the GUI, select the desired analysis from the pulldown menu and click the Start button. In this screenshot, a "Memory Error Analysis" is selected.**

For more complete information about compiler optimizations, see our Optimization Notice.



3 **Viewing the results of a deadlock and data race analysis in the Intel Inspector GUI**

Usage Example: Using a Host Pointer on the Device

The following code contains a memory problem:

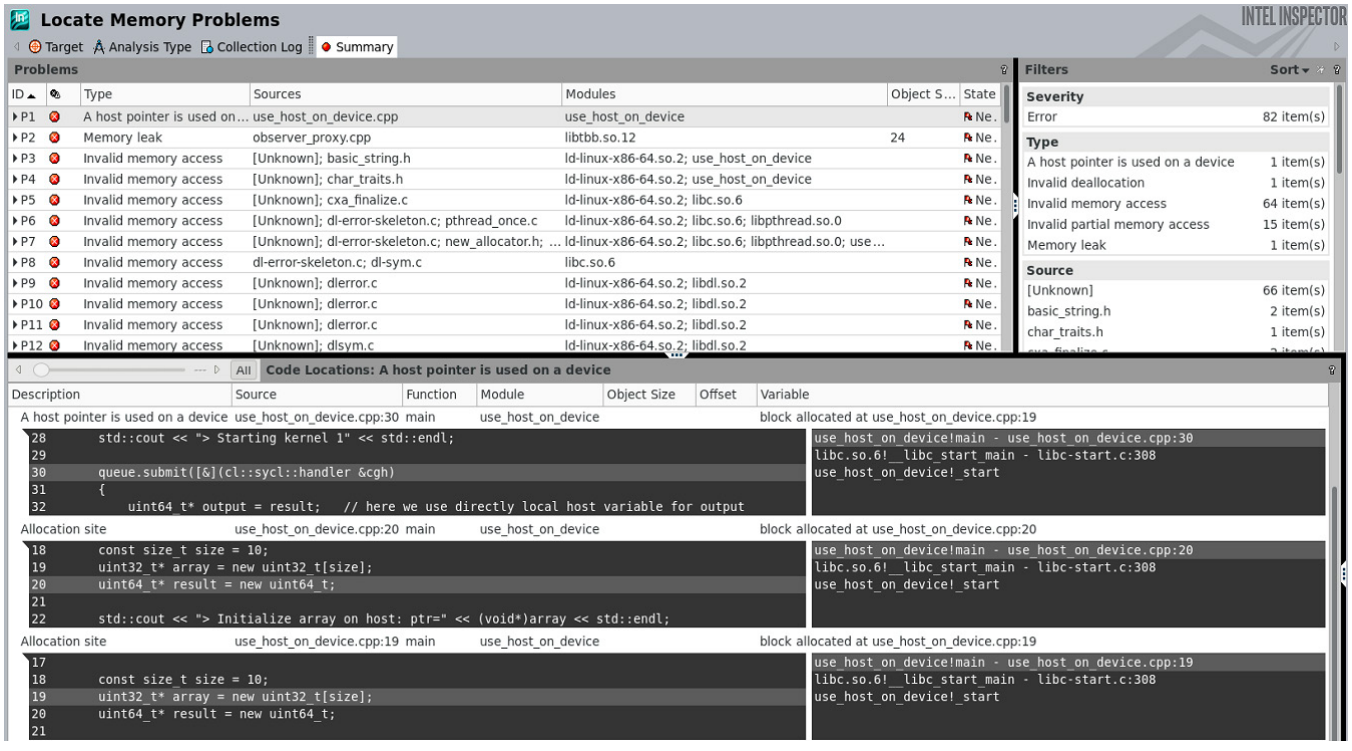
```
// … queue initialization code
const size_t size = 10;
uint32_t* array = new uint32_t[size];
uint64_t* result = new uint64_t;
// … array initialization code
queue.submit([&](cl::sycl::handler &cgh)
{
       uint64_t* output = result;   // here we use directly local host variable
for output
       cgh.parallel_for<class my_task>(cl::sycl::range<1> { size },
[=](cl::sycl::id<1> idx)
       {
              output[0] += array[idx];  // here we use directly the array from
host
       });
});
queue.wait();
```

Launch this code using the Intel Inspector command-line client and view the analysis results in the GUI (**Figure 4**):

```
$ inspxe-cl -c mi3 -- ./use_host_on_device

Collection started. To stop the collection, either press CTRL-C or enter from
another console window: inspxe-cl -r /tmp/mtutin/r000mi3 -command stop.
> RUNNING ON: Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz
> Initialize array on host: ptr=0x2e3d1c0
> Result: ptr=0x7ffc7a072310
> Starting kernel 1
> Done. Result=0x1a69f00
Elapsed: 116.516 sec

83 new problem(s) found
    1 A host pointer is used on a device problem(s) detected
    1 Invalid deallocation problem(s) detected
    64 Invalid memory access problem(s) detected
    15 Invalid partial memory access problem(s) detected
    2 Memory leak problem(s) detected
```



4 **Intel Inspector analysis showing the location of the memory error in the example code**

For more complete information about compiler optimizations, see our Optimization Notice.

Usage Example: Finding a Data Race

The following code contains a race condition:

```
void parallel_sum(uint32_t* input, size_t size, uint64_t* output)
{
    *output = 0;
    #pragma omp target device(0) map(to:input[0:size]) map(tofrom:output[0:1])
    #pragma omp parallel for
    for(size_t i=0;i<size;i++)
    {
        //#pragma omp atomic
        *output += input[i];
    }
}
```

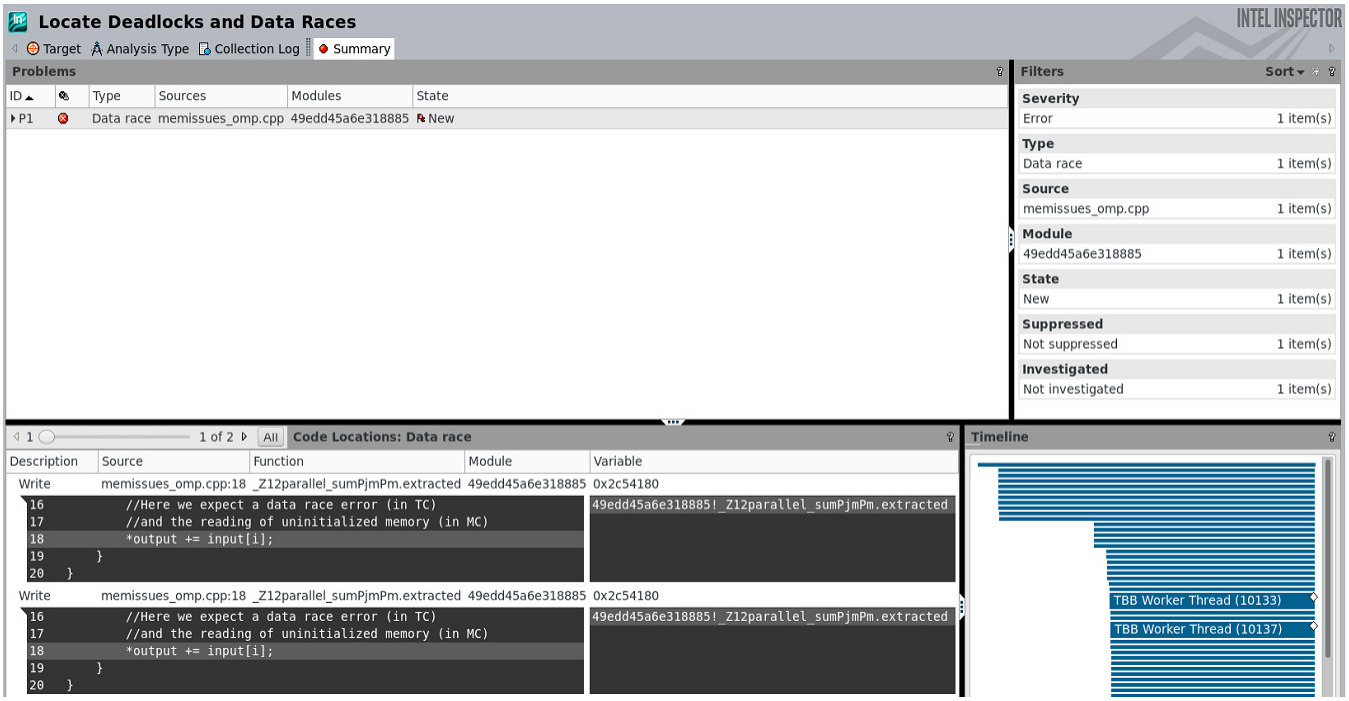
Launch this code using the Intel Inspector command-line client and view the analysis results in the GUI (**Figure 5**):

```
$ inspxe-cl -c ti3 -- ./memissues_omp_exe

Collection started. To stop the collection, either press CTRL-C or enter from
another console window: inspxe-cl -r /tmp/memissues_omp_exe/r002ti3 -command
stop.
> RUNNING ON: device no = 0
> Starting kernel
> output ptr = 0xbf4880
> Done. Result=433998319470 Expected=250000
Elapsed: 72.9886 sec

1 new problem(s) found
    1 Data race problem(s) detected
```

For more complete information about compiler optimizations, see our Optimization Notice.



5 **Intel Inspector analysis results showing a write-write data race in the example code**

NEWS HIGHLIGHTS

University of Illinois to Bring oneAPI Cross-Architecture Programming Model to NAMD

The Beckman Institute for Advanced Science and Technology at University of Illinois announced a new oneAPI Center of Excellence (CoE) to bring the oneAPI programming model to the life sciences application NAMD to additional heterogeneous computing environments. NAMD, which simulates large biomolecular systems, is helping to tackle real-world challenges such as COVID-19.

Read on >

For more complete information about compiler optimizations, see our Optimization Notice.

Table 1. Memory issues that Intel Inspector can detect

Problem Class	Example	Supports Early Interception?
Memory leak	`clBuff = clCreateBuffer(…)` `//clReleaseMemObject(clBuff)`	Yes

Invalid deallocation	`ptr = malloc(…)`
	`clBuff =`
	`clCreateBuffer(USE_HOST_PTR, ptr, …)`
	`//clReleaseMemObject(clBuff)`
	`free(ptr);`
		Yes

Uninitialized read	`ptr = malloc(…)`
	`clBuff =`
	`clCreateBuffer(USE_HOST_PTR, ptr, …)`
	
	`read of clBuff (in the kernel)`
		No

Invalid access	`ptr = malloc(…)`
	`clBuff =`
	`clCreateBuffer(USE_HOST_PTR, ptr, …)`
	`clReleaseMemObject(clBuff) and/or`
	`free(ptr)`
	`read\write of clBuff (in the kernel)`
		No

Missing allocation	`clReleaseMemObject(ptr) //release`
	`invalid handle`
	`//here OCL will throw an exception`
		Yes

Invalid arguments	`b1 = clCreateBuffer(NULL,`
	`CL_MEM_COPY_HOST_PTR)`
	`b2 = clCreateBuffer(random_ptr,`
	`CL_MEM_COPY_HOST_PTR)`
		Yes

Invalid buffer region	`uint32_t* array = new`
	`uint32_t[1024];`
	`cl::sycl::buffer<uint32_t, 1>`
	`inputBuf(array + 512, 1024);`
	`queue.submit([&](cl::sycl::handler`
	`&cgh) {`
		Yes

Table 2. Intel Inspector can detect data races.

Problem Class	Example

Data race	`cgh.parallel_for(…){`
	`  if(idx.get(0) > 0 && idx.get(0) <`
	`ArraySize - 1)`
	`  {`
	`  uint32_t s = array2_acc[idx-1] +`
	`array2_acc[idx] + array2_acc[idx +`
	`1];`
	`  array2_acc[idx] = (s / 3) %`
	`ArraySize;`
	`  }`
	`});`

For more complete information about compiler optimizations, see our Optimization Notice.

Conclusions

oneAPI provides a standard, simplified programming model that can run seamlessly on the scalar, vector, matrix, and spatial architectures deployed in CPUs and accelerators. It gives users and domain experts the much-needed freedom to focus on the code itself and not the underlying mechanism that generates the best possible machine instructions. Correctness analysis tools like Intel Inspector provide much-needed assistance in debugging difficult-to-detect threading and memory issues.

References

- **oneAPI**
- **Intel oneAPI Toolkits**
- **Intel® Inspector**

For more complete information about compiler optimizations, see our Optimization Notice.

For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
© Intel Corporation

Uncovering More Tuning Opportunities with Intel® Compiler Optimization Reports

Code Generation, Interprocedural Optimization, Floating-Point Precision, and More

Mayank Tiwari, Cloud Software Engineer, and Rama Malladi, Graphics Performance Engineer, Intel Corporation

The compiler reports generated by the Intel C, C++, and Fortran compilers provide useful information for optimizing code. Our previous articles in *The Parallel Universe* (see issues **41** and **42**) discussed compiler reports for loop transformations and vectorization. In this article, we'll cover code generation optimizations, interprocedural optimization (IPO), inlining, data alignment, OpenMP and auto-parallelization, and floating-point precision reports. These reports highlight the code generation done by the compiler and help us get better performance by doing "last mile" code changes and optimizations.

For more complete information about compiler optimizations, see our Optimization Notice.

Generating Optimization Reports

For the Intel® C/C++ compiler on Linux and macOS, the `-qopt-report[=n]` option requests an optimization report. Use the `/Qopt-report[:n]` option on Windows.) The 'n' is optional and indicates the level of detail in the report: 0 (no report) through 5 (detailed report). In this article, we'll discuss reports generated using `-qopt-report=5` (detailed report).

Code Generation

The code generation optimizations section of the compiler report can help in determining the number of hardware registers available, used, and spills, fills. The simple C++ vector addition shown in **Figure 1** gives the compiler optimization report shown in **Figure 2**. It shows detailed register usage for the input arguments, global and local variables, register spills, and stack usage.

```
vec01.cpp
17:    clock_t t1, t2;
18:
19:    t1 = clock();
20:    for(int k = 0; k < N_ITER; k++){
21:           for(int i = 0; i < size; i++)
22:                  crr[i] = arr[i] + brr[i];
23:    }
24:    t2 = clock();
25:
26:    cout << (double)(t2 - t1)/CLOCKS_PER_SEC << " seconds" << endl;
```

1 **Simple vector addition in C++**

Registers are the closest memory to the ALUs in a CPU, so reads and writes to registers are fast. For best performance, programmers should ensure that most of the variables in a routine can be accommodated in these registers. However, the number of registers is limited, so the compiler attempts to generate code for optimal allocation of these registers.

If a routine uses more variables than available registers, some variables may need to be stored and reloaded from the stack. This is known as register spilling. Sometimes it's unavoidable, but one can optimize register usage by applying loop optimizations like loop fission when possible. If the compiler optimization report indicates a significant number of spills, this is referred to as a high register pressure performance issue. Recommended optimizations for register pressure include avoiding loop unrolling, loop fission, and having the compiler generate scalar plus vector code.

For more complete information about compiler optimizations, see our Optimization Notice.

```
Hardware registers
        Reserved     :    3[ rsp r13 rip]
        Available    :   38[ rax rdx rcx rbx rbp rsi rdi r8-r12 r14-r15 mm0-mm7
zmm0-zmm15]
        Callee-save  :   17[ rbx rbp rsi rdi r12 r14-r15 xmm6-xmm15]
        Assigned     :   11[ rax rdx rcx rsi r8 zmm0-zmm5]

    Routine temporaries
        Total         :      59
            Global    :      28
            Local     :      31
        Regenerable   :      18
        Spilled       :       1

    Routine stack
        Variables     :   16435 bytes*
            Reads     :       3 [7.30e-008 ~ 0.0%]
            Writes    :       4 [9.75e-008 ~ 0.0%]
        Spills        :       0 bytes*
            Reads     :       0 [0.00e+000 ~ 0.0%]
            Writes    :       0 [0.00e+000 ~ 0.0%]
```

2 **Intel® C++ compiler code-generation report showing register pressure and stack usage**

Interprocedural Optimization

Take a look at the pseudocode shown in **Figure 3**. It was adapted from a Lattice Boltzmann Method (LBM) application. The compiler optimization reports with and without IPO are shown in **Figures 4** and **5**. The compilation units, `lbm_file1.c` and `lbm_file2.c`, contain functions `func_grids` and `func2`, respectively. `func2` invokes `func_grids` in a time-step loop.

One of the most common optimizations for better compiler code generation and performance is function inlining, but it can be done only when the callee (function definition) and caller (invocation) are in a single compilation unit. This isn't the case for the LBM example in **Figure 3**. However, the Intel compiler can do interprocedural optimization (the `-ipo` and `/Qipo` compiler options on Linux and Windows, respectively). As shown in **Figure 4**, the call to `func_grids` prevents vectorization of the loop in `func2`. Using IPO gives the compiler an opportunity to vectorize such loops/function calls by inline optimization (**Figure 5**).

For more complete information about compiler optimizations, see our Optimization Notice.

```
lbm_file1.c
11 void func_grids(...) {
12     ...
13     ...
18 }

lbm_file2.c
82 int func2(...) {
83   ...
80   for(t = 1; t <= timeSteps; t++ ) {
81     ...
84     func_grids(...);
85   }
86   ...
98 }
```

3 **Example code showing the use of IPO compiler optimization**

```
lbm_file1.optrpt
Begin optimization report for: func_grids(...)
...
lbm_file2.optrpt
  -> EXTERN: (84,3) func_grids(...)
...
LOOP BEGIN at lbm_file2.c(80,2)
...
  remark #15382: vectorization support:
    call to function func_grids(...) cannot be vectorized [ func2.c(84,3) ]
```

4 **Compiler optimization reports for each file without IPO**

For more complete information about compiler optimizations, see our Optimization Notice.

```
ipo_out.optrpt
INLINE REPORT: (func2(...))
...
-> INLINE: (84,3) func_grids(...) (isz = 6) (sz = 13)
...
LOOP BEGIN at lbm_file2.c(80.2)
...
remark #15301: PARTIAL LOOP WAS VECTORIZED
...
=================================================
Begin optimization report for: func_grids(...)
    Report from: Interprocedural optimizations [ipo]
DEAD STATIC FUNCTION: (func_grids(...)) lbm_file1.c
=================================================
```

5 **Compiler optimization report using IPO showing inlining, vectorization, and dead static function elimination. Note that only one report (`ipo_out.optrpt`) is generated.**

Function Inlining

One of the most common compiler optimizations is to inline a called function within the caller. The smaller a function's size, the more likely it is to be inlined. The compiler generally tries to limit the "code bloat" caused by inlining, but users can override the compiler's conservative tendencies by changing the options listed in **Figure 6**.

```
INLINING OPTION VALUES:
  -Qinline-factor: 100
  -Qinline-min-size: 30
  -Qinline-max-size: 230
  -Qinline-max-total-size: 2000
  -Qinline-max-per-routine: 10000
  -Qinline-max-per-compile: 500000
```

6 **Compiler options and heuristics for inlining**

For more complete information about compiler optimizations, see our Optimization Notice.

Floating-Point Model and Precision

Scientific users usually want to maintain high precision in their computations. This is typically accomplished by using 64-bit instead of 32-bit floating-point datatypes. However, using lower precision datatypes when appropriate can improve performance. In addition, compiler optimizations that affect numerical reproducibility/consistency are sometimes prevented using the `-fp-model precise` option (`fp:precise` on Windows).

The code in **Figure 7** computes the square norm of a 2D array. If this code is compiled with the precise floating-point model, the compiler is unable to vectorize the inner loop (**Figure 8**). A more relaxed floating-point model (the default) allows the compiler to do more aggressive optimization, as suggested in the report.

```
file5.f
35   do j=1,Y
36     do i=1,X
37       norm = norm + Q(i,j)*Q(i,j)
38     enddo
39   enddo
```

7 **A simple two-level reduction loop**

```
LOOP BEGIN at file5.f(36,5)
remark #15331: loop was not vectorized: precise FP model implied by the command
line or a directive prevents vectorization. Consider using fast FP model
```

8 **A precise floating-point model can limit optimization**

OpenMP and Auto-Parallelization

Another speedup opportunity on modern processors is parallelism. The Intel compilers support parallelism when the compiler can determine that it is safe (auto-parallelization) or when the parallelism is expressed using OpenMP (**Figure 9**). The compiler optimization reports which loops are parallelized when the OpenMP (`-qopenmp` and `/Qopenmp` on Linux and Windows, respectively) and auto-parallelization (`-parallel` and `/Qpar` on Linux and Windows, respectively) options are used (**Figure 10**).

For more complete information about compiler optimizations, see our Optimization Notice.

file_par.c

```
23 #pragma omp parallel for private(p) num_threads(nthr)
24   for (p=0; p < size; p++)
25   {
26      ...
86   }

129 for(z = 0; z < SIZE_Z; z++)
130   for(y = 0; y < SIZE_Y; y++)
131     for(x = 0; x < SIZE_X; x++)
132     {
133        ...
151     }
```

9 **Code snippet with OpenMP parallel constructs and potential compiler auto-parallelization**

```
Report from: OpenMP optimizations [openmp]
   OpenMP Construct at file_par.c(23,1)
   remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED

LOOP BEGIN at file_par.c(129,1)
   remark #25101: Loop Interchange not done due to: Original Order seems proper
   remark #25452: Original Order found to be proper, but by a close margin
   remark #17109: LOOP WAS AUTO-PARALLELIZED
```

10 **Compiler report showing OpenMP and auto-parallelization**

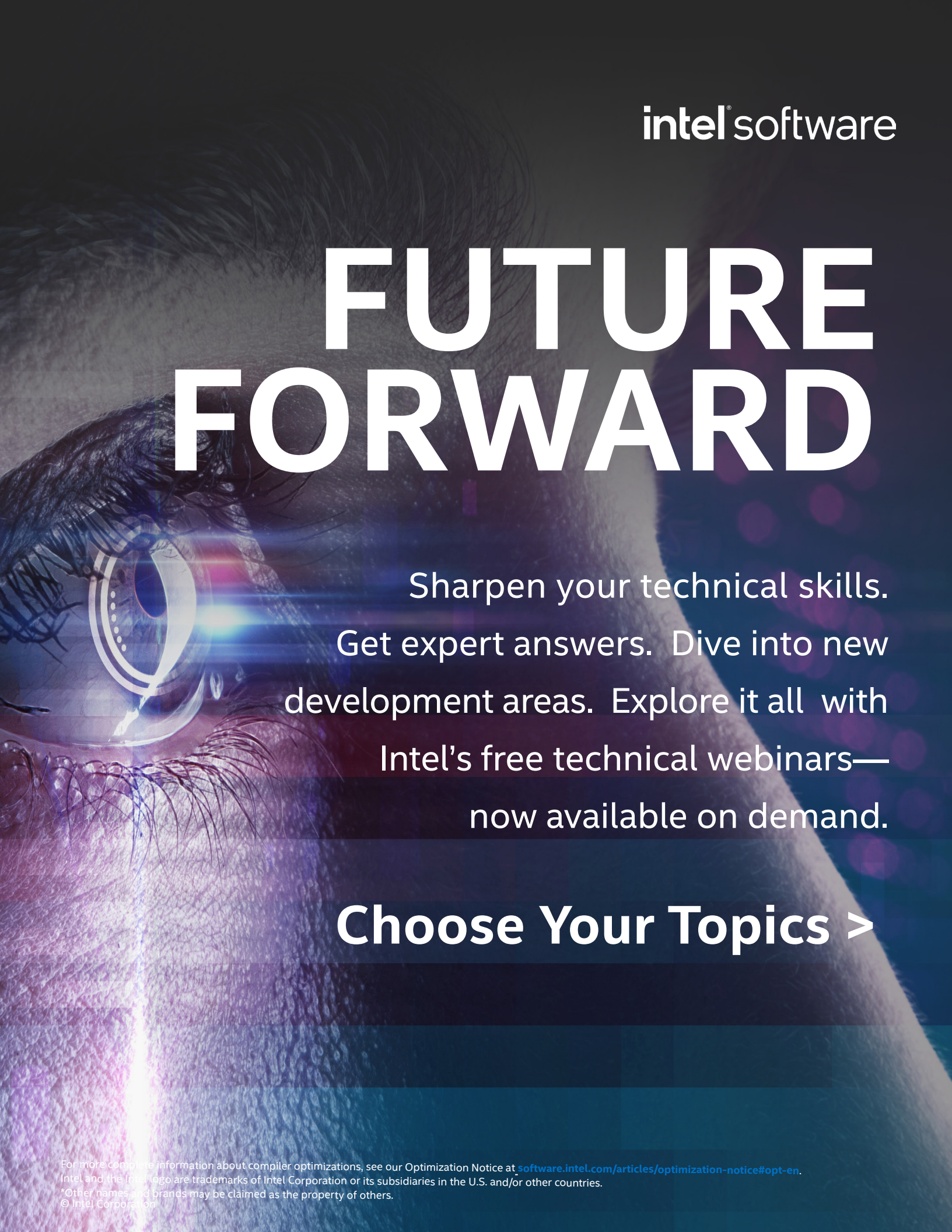
Closing Comments

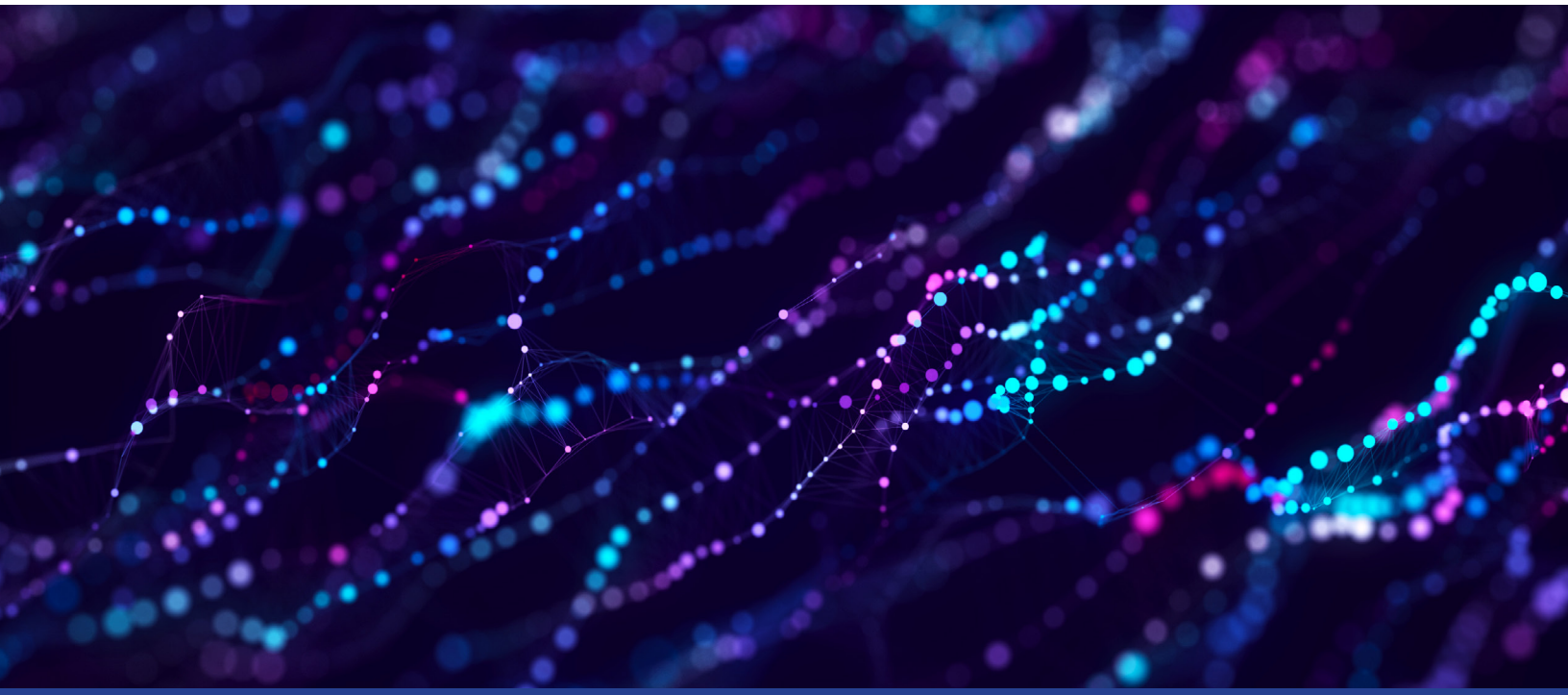
The Intel compilers provide a rich set of features, performance optimizations, and support for the latest language standards. Users are encouraged to try the latest compiler and experience application performance improvements that are possible by changing a few compiler options. The compiler reports discussed in this and previous articles help to understand what the compiler is doing. The examples shown in these articles helps drive this point home.

Learn More

1. **Intel® 64 and IA-32 Architectures Software Developer Manuals**
2. **Intel® C++ Compiler Classic Developer Guide and Reference**
3. **Consistency of Floating-Point Results using the Intel® Compiler**

For more complete information about compiler optimizations, see our Optimization Notice.


intel software
FUTURE FORWARD
Sharpen your technical skills. Get expert answers. Dive into new development areas. Explore it all with Intel's free technical webinars—now available on demand.
Choose Your Topics >
For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.
© Intel Corporation



Cluster-Wide MPI Tuning Using Intel® MPI Library

Tune MPI Collective Communication with the `mpitune_fast` Utility

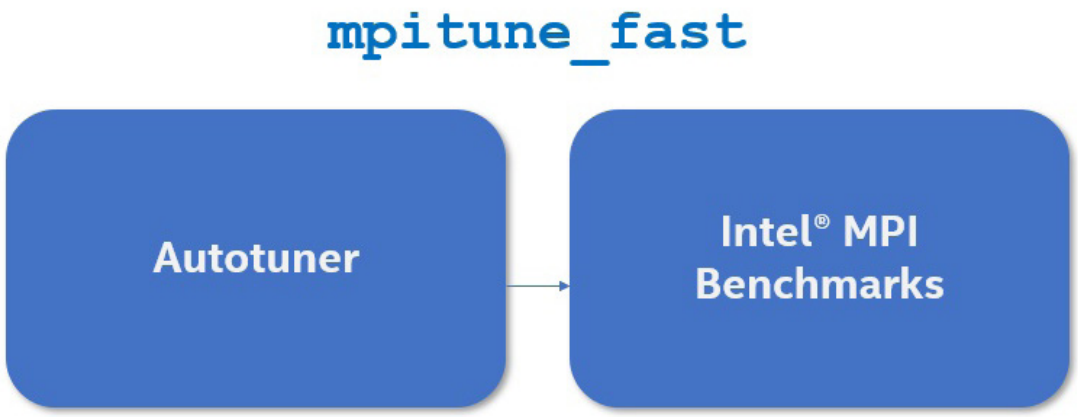
Dr. Amarpal S Kapoor, Technical Consulting Engineer, and Marat Shamshetdinov, Software Development Engineer, Intel Corporation

This article continues our series on the tuning utilities in **Intel® MPI Library**. While the previous articles primarily focused on application-specific tuning tools and methodologies, this one focuses on cluster-wide tuning using a utility called `mpitune_fast`. Ideally, `mpitune_fast` should be run by a cluster administrator to ensure that users get an optimally tuned configuration of Intel MPI Library. However, `mpitune_fast` can be run by unprivileged users at any time. This article introduces `mpitune_fast` and describes a tuning methodology and the resulting performance gains for the **Intel® MPI Benchmarks**. All experiments in this article use Intel MPI Library 2019 U9. Results are presented from two Intel® Xeon® processor-based clusters: **Endeavor** and the **Intel® DevCloud**.

For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

The ease of use, low overhead, and potential performance gains of **Autotuner** inspired Intel MPI Library's development team to extend its scope beyond application-specific tuning to cluster-wide tuning. IMB is used generate tuning data that is generalizable to most MPI applications. Combining Autotuner and IMB resulted in a cluster-wide tuning utility called `mpitune_fast` (**Figure 1**).



1 **Components of `mpitune_fast`**

`mpitune_fast` iteratively runs IMB with predefined settings to generate cluster-specific tuning parameters that are better than Intel MPI Library's default settings. The resulting configuration is stored in a file that can be used by all MPI applications running on the cluster. Cluster administrators can set the `I_MPI_TUNING_BIN` environment variable to point to this file so that all MPI applications running on the cluster can benefit from the `mpitune_fast` analysis.

Key Features of `mpitune_fast`

Cluster-Wide Tuning

Cluster-wide tuning refers to two capabilities:

1. **Generation of tuning data** that remains valid for any application running on the cluster
2. **Generation of tuning settings** that combine multiple nodes (1 to N_{max}) and processes per node (1 to C_{max})

Consequently, `mpitune_fast` only needs to be run once as long as there are no changes to the cluster configuration.

Dynamic Tuning

`mpitune_fast` is based on Autotuner, so it inherits dynamic tuning capabilities, which greatly reduces the overall tuning overhead and simplifies user workflows.

For more complete information about compiler optimizations, see our Optimization Notice.

Parallel Tuning

`mpitune_fast` is a cluster-wide tuning tool, so it's important to tune for multiple MPI rank placement schemes and a variable number of nodes. This is achieved by tuning in parallel for multiple values of processes per node (`-ppn`) and total number of nodes (`-n`), whenever possible. When tuning for high values of `-ppn` (tending towards the number of physical cores per node) and `-n` (tending towards the total number of nodes in the hostfile), the tuning runs are inherently serial. However, for smaller `-ppn` and `-n` values, `mpitune_fast` automatically launches parallel tuning instances to better utilize the hardware and reduce the overall tuning overhead.

Ease of Use

A design goal of `mpitune_fast` is to maintain a simple user workflow and invocation scheme. Therefore, any complexity associated with running IMB and Autotuner is hidden from the users. `mpitune_fast` carefully configures underlying tools through runtime options and environment variables.

Methodology

Using `mpitune_fast` is very simple. The following command launches `mpitune_fast` on clusters running the LSF or Slurm job schedulers (automatic detection of hostfile is enabled):

```
$ mpitune_fast
```

For clusters running other job schedulers, additionally a file containing the list of nodes on which to run `mpitune_fast` must to be specified:

```
$ mpitune_fast -f ./hostfile
```

`mpitune_fast` only has a handful of arguments. They can be viewed using the following help option,

```
$ mpitune_fast -h
```

By default, `mpitune_fast` tunes for multiple processes per node (`-ppn`) and number of nodes (`-n`) (i.e., all powers of two up to the physical core count including the physical core count for `-ppn` and all powers of two up to the host count). For example, for a cluster with 50 nodes and 24 physical cores per node, by default `mpitune_fast` would test `-n` values of 1, 2, 4 , 8, 16, 32, and 50 and `-ppn` values of 1, 2, 4, 8, 16, and 24. If common usage patterns for `-n` and `-ppn` used by jobs on a cluster are known, one may optionally limit the `-ppn` and `-n` values to one or more values (in addition to specifying custom values) by a comma-separated list:

For more complete information about compiler optimizations, see our Optimization Notice.

```
$ mpitune_fast -ppn 24,12,6 -n 50,25 -f ./hostfile
```

Tuning overhead may be further reduced by restricting the scope of collectives to tune using the `-c` option (`allreduce`, `reduce`, `bcast`, and `barrier` are tuned by default):

```
$ mpitune_fast -ppn 24,12,6 -n 50,25 -f ./hostfile -c allreduce,barrier
```

Running these commands generates a tuning file that can be used as the default tuning file for all MPI applications running on the cluster in question. One must be careful when using the `-n` option to `mpitune_fast`. Here, it represents the number of nodes. In the context of `mpirun` or `mpiexec.hydra`, `-n` represents the total number of ranks. `mpitune_fast` also accepts the `-d` option to store the tuning results in a user-specified directory.

To evaluate the performance benefits of `mpitune_fast`, we use IMB to measure the performance of common MPI collective communication functions: `Allreduce`, `Bcast`, `Reduce`, `Scatter`, and `Gather`. Our test script performed three main steps.

First, IMB was run with Intel MPI Library's default tuning configuration using the following command:

```
$ mpiexec.hydra -bootstrap ssh -n $NRANKS -ppn $PPN -f hostfile
IMB-MPI1 $LIST -npmin $NRANKS -iter $REP_LARGE -iter_policy off
-msglog 0:16 -mem $MEM_LIMIT -time $TIME_LIMIT -imb_barrier 1
```

Table 1 shows the values of variables used for this command.

Table 1. Test configuration

Variable	Value	
	Endeavor	DevCloud
NRANKS	768	
PPN	48	
NODES	16	
LIST	allreduce,bcast,reduce,scatter,gather	
REP_LARGE	1,000	
MEM_LIMIT	999,999	
TIME_LIMIT	100,000	

For more complete information about compiler optimizations, see our Optimization Notice.

To have tighter control over the number of repetitions to run per message size in IMB, instead of running a single command to test performance over the entire 1 B to 4 MB message range, we chose to split it into three separate executions. The arguments to `-iter` and `-msglog` changed across the three executions as shown in **Table 2**.

Table 2. Message size driven repetition selection logic

-iter	-msglog
1,000	1:16
600	17:19
20	20:22

As shown in **Table 2**, we performed 1,000 repetitions for small messages of sizes 2^1 B to 2^{16} B, 600 repetitions for medium messages of sizes 2^{17} B to 2^{19} B, and 20 repetitions for large messages of sizes 2^{20} B to 2^{22} B.

The performance data collected from this step served as baseline performance. We chose to track IMB's `t_max` metric per message size, which is the worst observed performance across all ranks in a collective call, and therefore a safe performance measure.

We then ran `mpitune_fast` as an unprivileged user (to extend the scope of this article to all cluster users, not just cluster administrators):

```
$ mpitune_fast -f hostfile -ppn $PPN -n $NNODES -c $LIST
```

We restricted tuning to a specified number of nodes, number of processes per node, and MPI functions of interest. This step generated a binary file with tuned settings.

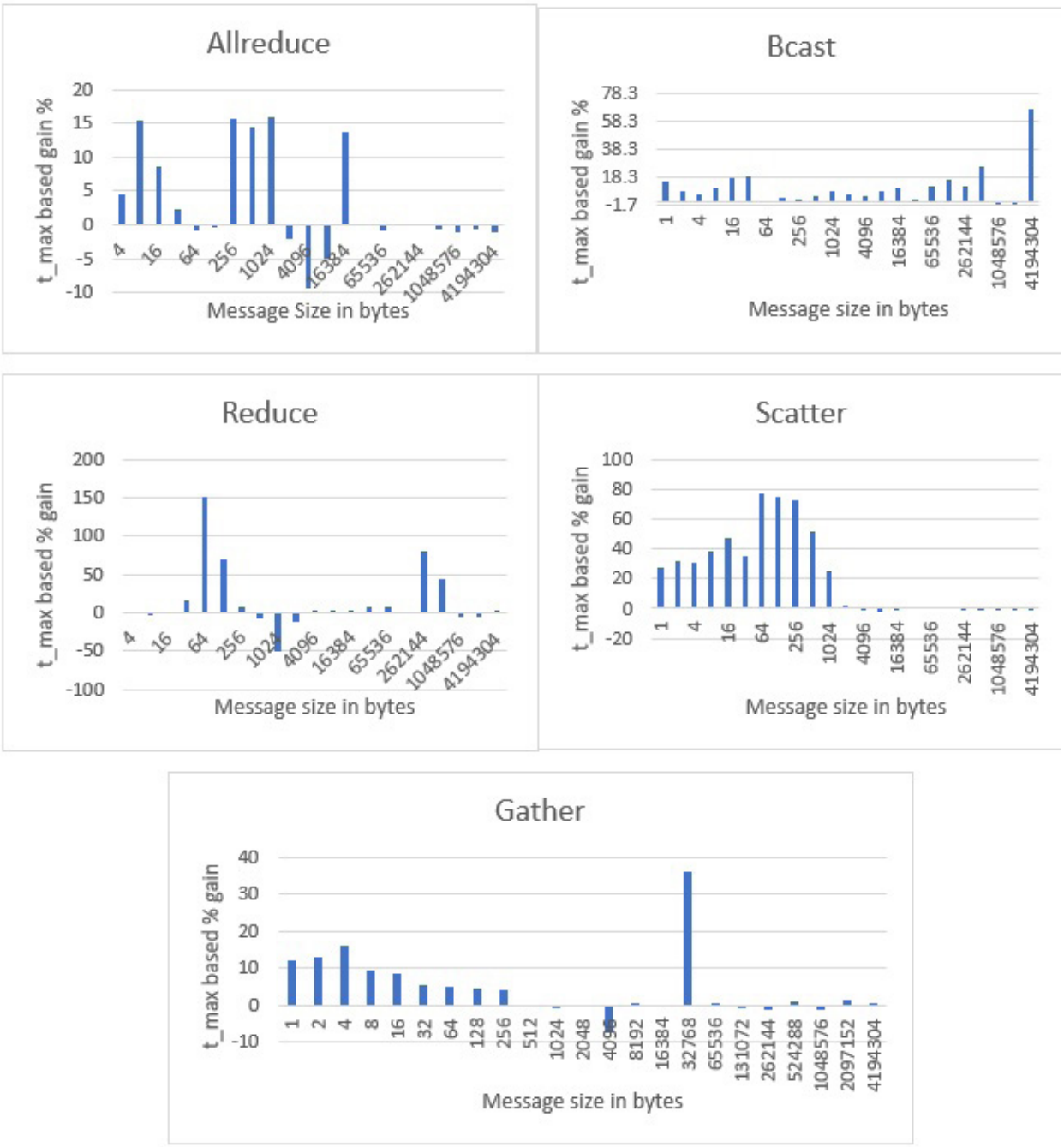
Finally, the tuning file was used to assess the benefit of the `mpitune_fast` analysis. The following environment variable directs Intel MPI Library to use a specified tuning file:

```
$export I_MPI_TUNING_BIN=./cluster_merged<postfix.dat>
```

For more complete information about compiler optimizations, see our Optimization Notice.

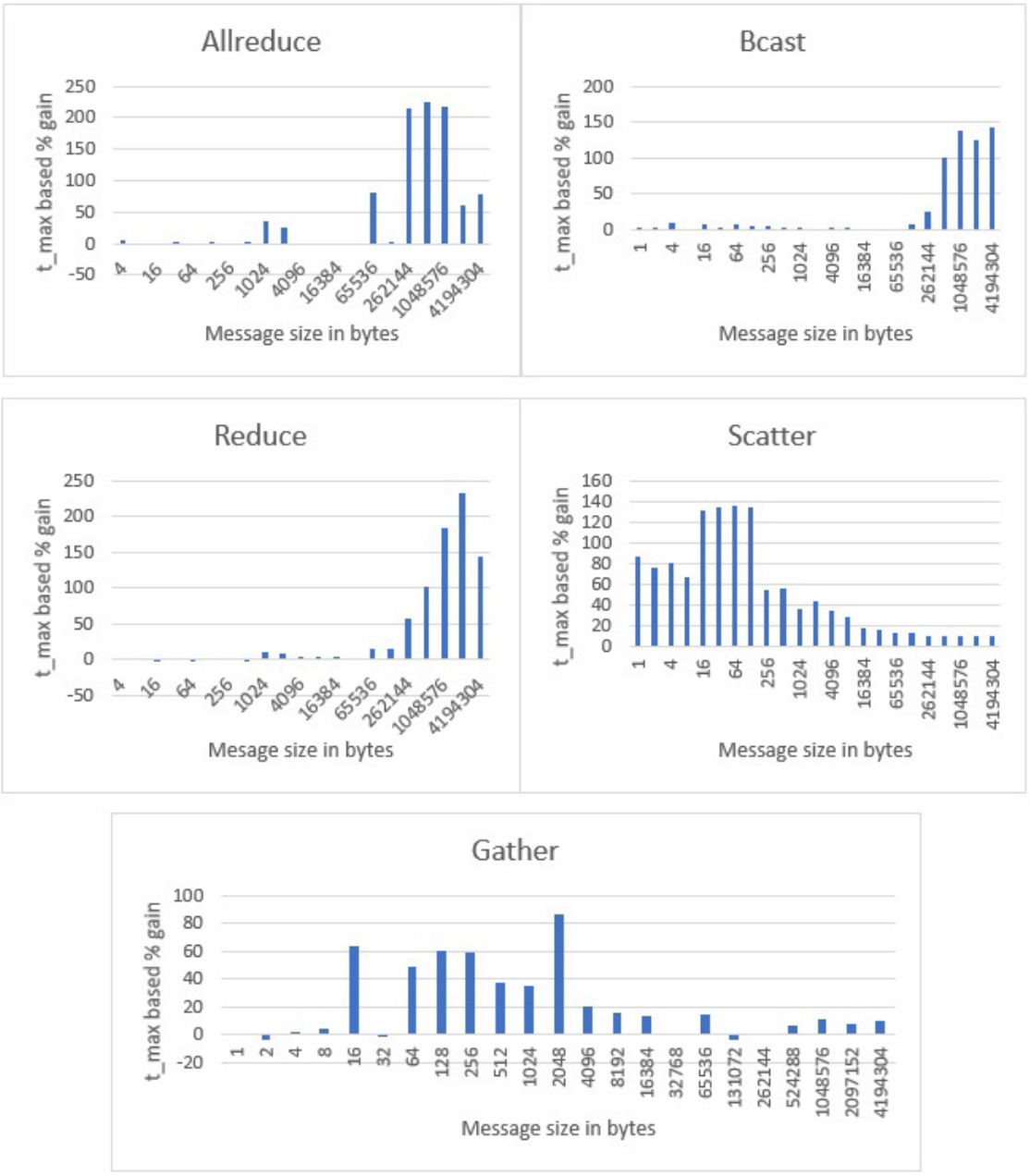
Results

This section presents the data we collected from running the steps described in the previous section on two Intel® Xeon® processor-based clusters: Endeavour and Intel DevCloud. On Endeavour, we used 16 Intel Xeon Platinum 8268 processor dual-socket nodes connected through a Mellanox Quantum HDR interconnect. On Intel DevCloud, we used eight Intel Xeon Gold 6128 processor dual-socket nodes connected through an Ethernet interconnect.



2 Performance improvement for five common MPI collective communication functions on Endeavour

For more complete information about compiler optimizations, see our Optimization Notice.



3 **Performance improvement for five common MPI collective communication functions on the Intel DevCloud**

Figure 2 shows the performance gains we observed on Endeavour. An average gain of 11.15% was observed for all functions over the entire range of message sizes. Out of the 111 data points shown in **Figure 2**, five data points show some performance degradation. `mpitune_fast` developers are working on refining the tuning methodologies to eliminate such behavior. Also, such minor degradations may be attributed to noise and network traffic coming from other applications running on nodes connected to the same switch.

For more complete information about compiler optimizations, see our Optimization Notice.

Figure 3 shows the performance gains observed on Intel DevCloud. An average gain of 36.35% was observed for all functions over the entire range of message sizes. Unlike Endeavour, no performance degradations were observed.

Limitations

The 2019 U9 version of `mpitune_fast` has the following limitations:

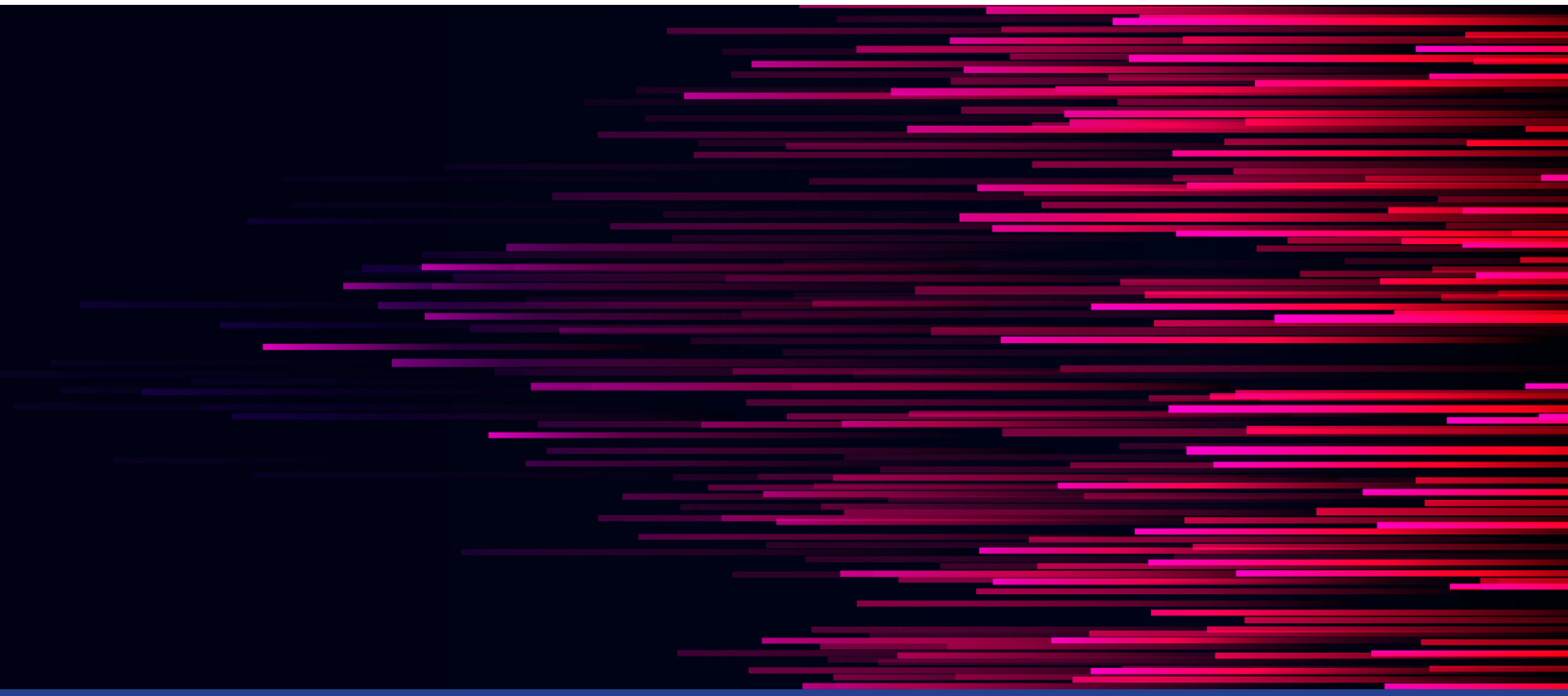
1. **Non-blocking collectives** aren't supported yet.
2. **By design, `mpitune_fast` currently only supports IMB to generate tuning data.** Tuning based on user-specified benchmark applications is not supported.
3. **Conditional tuning** specific to user-defined message sizes is not currently available.

Summary

This article introduced `mpitune_fast`, one of Intel MPI Library's tuning utilities, to conveniently generate cluster-wide tuning data. Both cluster administrators and unprivileged users can run this utility. Average gains of 11.15% and 36.35% were observed for five common MPI functions on Endeavour and Intel DevCloud, respectively. While the tuning data generated by `mpitune_fast` is applicable to any application, Autotuner can generate application-specific tuning data, thereby providing additional performance gains in user applications. The recommended workflow is for cluster administrators to first run `mpitune_fast` to generate optimal Intel MPI Library tuning settings for their clusters. Cluster users can then run Autotuner to generate even better settings for their applications.

For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues



Accelerating Linear Models for Machine Learning

Linear Regression Has Never Been Faster

Victoriya Fedotova, Machine Learning Engineer, Intel Corporation

If you've ever used Python and scikit-learn to build machine learning (ML) models from large data sets, you may have also wished that you could make these computations go faster. What if I told you that altering a single line of code could accelerate your ML computations? What if I also told you that getting faster results doesn't require specialized hardware?

For more complete information about compiler optimizations, see our Optimization Notice.

In this article, I'll teach you how to train ridge regression models using a **version of scikit-learn that's optimized for Intel CPUs**, then compare the performance and accuracy of these models trained with the vanilla scikit-learn library. This article continues our series on accelerated ML algorithms.

- **Fast Gradient Boosting Tree Inference for Intel® Xeon® Processors**
- **K-means Acceleration with 2nd Generation Intel® Xeon® Scalable Processors**

A Practical Example of Linear Regression

Linear regression, a special case of ridge regression, has a lot of real-world applications. For my comparisons, I'm going to use the well-known **House Sales in King County, USA data set** from **Kaggle.** This data set is used to predict house prices based on one year of King County sales data.

This data set has 21,613 rows and 21 columns. Each row represents a house that was sold in King County between May 2014 and May 2015. The first column contains a unique identifier for the sale, the second column contains the date the house was sold, and the third column contains the sale price, which is also the target variable. Columns 4 to 21 contain various numerical characteristics of the house, such as the number of bedrooms, square footage, the year when the house was built, zip code, etc. I am going to build a ridge regression model that predicts the price of the house based on the data in columns 4 to 21.

In theory, the coefficients of the linear regression model should have the lowest residual sum of squares (RSS). In practice, the model with the lowest RSS is not always the best. Linear regression can produce inaccurate models if input data suffers from multicollinearity. Ridge regression can give more reliable estimates in this case.

Solving a Regression Problem with scikit-learn

Let's see how to build a model with `sklearn.linear_model.Ridge`. The program below trains a ridge regression model on 80% of the rows from the House Sales dataset, then uses the other 20% to test the model's accuracy.

For more complete information about compiler optimizations, see our Optimization Notice.

```
import numpy as np
import pandas as pd
from sklearn import config_context
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split

# Load the data from a text file into pandas DataFrames.
# The third column contains the responses. Load it into df_y.
# Columns after the fourth contain features. Load them into df_X.

infile = "data/kc_house_data.csv"
df_X = pd.read_csv(infile, usecols=range(3, 20))
df_y = pd.read_csv(infile, usecols=[2])

# Split the data into training and testing subsets.
# Use 80% of the rows to train the regression model.
# Use the remaining 20% to test the model.

X_train_np, X_test_np, y_train_np, y_test_np = train_test_split(df_X, df_y, \
    train_size=0.8, random_state=7777777)

# Convert the data into pandas DataFrames for convenience:

X_train = pd.DataFrame(X_train_np)
y_train = pd.DataFrame(y_train_np)

X_test = pd.DataFrame(X_test_np)
y_test = pd.DataFrame(y_test_np)

# Construct a ridge regression algorithm object:

alg = Ridge(alpha=1.0)

# Train a ridge regression model assuming no infinities in the data:

with config_context(assume_finite=True):
    model = alg.fit(X_train, y_train)

# Check the model's accuracy on the test set:

y_pred = model.predict(X_test)

MSE = ((y_test.values - y_pred)**2).sum()
RMSD = np.sqrt(MSE / y_test.size)
R2 = alg.score(X_test, y_test)
```

For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

The resulting R^2 equals 0.69, meaning that our model describes 69% of variance in the data. See the Appendix for details on the quality of the trained models.

Intel-Optimized scikit-learn

Even though ridge regression is quite fast in terms of training and prediction time, you usually need to perform multiple training experiments to tune the hyperparameters. You might also want to experiment with feature selection (i.e., evaluate the model on various subsets of features) to get better prediction accuracy. Since one round of training can take several minutes on large data sets, which can quickly add up if your task requires multiple rounds of training, the performance of linear model training is critical.

Intel® Distribution for Python is a drop-in replacement for the native Python installation, but it's optimized for Intel® architectures. It works seamlessly with the packages available for installation through common channels such as conda and pip. The scikit-learn in Intel Distribution for Python has the same set of algorithms and the APIs as Continuum's scikit-learn, so no code changes are required to get a performance boost for ML algorithms.

Also, with **daal4py**, a Python interface to the **Intel® oneAPI Data Analytics Library**, it's possible to improve the performance of scikit-learn even further. daal4py provides configurable ML kernels, some of which support streaming input data and can easily be scaled out to clusters of workstations.

How to Configure scikit-learn with daal4py

There are several ways to install Intel® Distribution for Python. Follow **these instructions** to install it with conda.

```
conda install -n idp daal4py
```

Dynamic patching of scikit-learn is required to use daal4py as the underlying solver. You can enable patching without modifying your application. Just use the following command-line flag:

```
python -m daal4py my_app.py
```

Patching also can be enabled within your application:

```
import daal4py.sklearn
daal4py.sklearn.patch_sklearn()
```

For more complete information about compiler optimizations, see our Optimization Notice.

To undo the patch, run:

```
daal4py.sklearn.unpatch_sklearn()
```

Applying the patch impacts the following scikit-learn algorithms:

- **sklearn.linear_model.LinearRegression**
- **sklearn.linear_model.Ridge** (solver='auto')
- **sklearn.linear_model.LogisticRegression** and **sklearn.linear_model.LogisticRegressionCV** (solver in ['lbfgs', 'newton-cg'])
- **sklearn.decomposition.PCA** (svd_solver='full' and introduces svd_solver='daal')
- **sklearn.cluster.KMeans** (algo='full')
- **sklearn.metric.pairwise_distance** with metric='cosine' or metric='correlation'
- **sklearn.svm.SVC**

This list will continue to grow in the next releases of Intel Distribution for Python.

Performance Comparison

To compare performance of the vanilla scikit-learn and Intel-optimized scikit-learn, we used the King County data set plus six artificially generated datasets with varying numbers of samples and features. The latter were generated using the scikit-learn `make_regression` function:

```
X, y = make_regression(n_samples=nrows, n_features=ncols,
                       n_informative=ncols, noise=10.0, bias=10.0,
                       random_state=7777777)
X_train = pd.DataFrame(X)
y_train = pd.DataFrame(y)
```

Figure 1 shows the wall-clock time spent on training a ridge regression model with two different configurations:

- **Scikit-learn version 0.22** installed from the default set of conda channels.
- **Scikit-learn version 0.21.3** from Intel Distribution for Python optimized with daal4py.

To enable oneDAL optimizations in scikit-learn, we used the `-m daal4py` command-line option. For performance measurements, we used the **Amazon Web Services Elastic Compute Cloud (AWS EC2)**. We chose the instance that gives best performance:

- **CPU: c5.**metal (2nd Generation Intel Xeon Scalable processors, two sockets, 24 cores per socket)

Amazon states that "C5 instances offer the lowest price per vCPU in the Amazon EC2 family and are ideal for running advanced compute-intensive workloads." The c5.metal instance has the most CPU cores and the latest CPUs among all C5 instances.

For more complete information about compiler optimizations, see our Optimization Notice.

Data set Number of rows Number of features	House 17290 17	artificial 10000 100	artificial 10000 1000	artificial 100000 100	artificial 100000 1000	artificial 1000000 100	artificial 1000000 1000
Vanilla sklearn	2.54	3.57	47.12	23.53	359.45	132.04	3144.44
Intel-optimized sklearn	1.77	2.78	82.37	5.53	153.17	24.07	983.65
Vanilla Sklearn / Intel-optimized Sklearn	1.44	1.28	0.57	4.26	2.35	5.49	3.20

1 **Ridge regression training time**

See the Configuration section below for hardware details. See the Appendix for details on the quality of trained models.

Figure 1 shows that:

- **Ridge regression training is up to 5.49x faster** with the Intel-optimized scikit-learn than with vanilla scikit-learn.
- **The performance improvement from the Intel-optimized scikit-learn increases** with the size of the data set.

What Makes scikit-learn in Intel Distribution for Python Faster?

For big data sets, ridge regression spends most of its compute time on matrix multiplication. oneDAL's implementation of ridge regression relies on the **Intel® Math Kernel Library (Intel® MKL)**, which is highly optimized for Intel CPUs. Intel MKL uses Single Instruction Multiple Data (SIMD) vector instructions from the **Intel® Advanced Vector Extensions 512 (Intel® AVX-512)** available on 2nd Generation Intel Xeon Scalable processors. Compute-intensive kernels like matrix multiplication benefit significantly from the data parallelism that these instructions

For more complete information about compiler optimizations, see our Optimization Notice.

provide. Another level of parallelism for matrix multiplication is achieved by splitting matrices into blocks and processing them in parallel using Threading Building Blocks (TBB).

The Intel-optimized version of scikit-learn gives significantly better performance for ridge regression with no loss of model accuracy and little to no code modification. The performance advantages are not limited to just this algorithm. As mentioned previously, the list of optimized ML algorithms continues to grow.

Configuration

Hardware

c5.metal AWS EC2 instance: Intel Xeon 8275CL processor, two sockets with 24 cores per socket, 192 GB RAM. OS: Ubuntu 18.04.3 LTS.

Software

Vanilla sklearn: Python 3.8.0, scikit-learn 0.22, pandas 0.25.3.

Intel Distribution for Python scikit-learn: conda installation from the Intel channel: Python 3.7.4, scikit-learn 0.21.3 optimized with daal4py 2020.0 build py37ha68da19_8, pandas 0.25.1.

Python and accompanying libraries are the default versions installed by the conda package manager when configuring the respective environments.

Appendix

Table 1. Root mean squared deviation (RMSD) and the coefficient of determination (R^2) for ridge regression models

Ridge Regression Model Accuracy						
Data Set	Number of rows	Number of columns	RMSD		R^2	
			Vanilla scikit-learn	Intel-optimized scikit-learn	Vanilla scikit-learn	Intel-optimized scikit-learn
King County	17,290	17	200,221.014	200,221.014	0.69414	0.69414
make_regression	10,000	100	10.018	10.018	0.99969	0.99969
make_regression	10,000	1,000	9.609	9.609	0.99997	0.99997
make_regression	100,000	100	9.996	9.996	0.99970	0.99970
make_regression	100,000	1,000	9.987	9.987	0.99997	0.99997
make_regression	1,000,000	100	10.014	10.014	0.99969	0.99969
make_regression	1,000,000	1,000	9.995	9.995	0.99997	0.99997

For more complete information about compiler optimizations, see our Optimization Notice.

intel software

Intel® DevCloud

A Development Sandbox for Data Center to Edge Workloads

Develop, Test, and Run Your Workloads on a Cluster of the Latest Intel® Hardware and Software.

With integrated Intel® optimized frameworks, tools, and libraries, you'll have everything you need for your projects.

Try Out a Diverse Collection of Intel® Hardware.

Expand your skills and experiment with this state-of-the-art cluster that offers capabilities like natural language processing and time-series analysis—plus edge acceleration hardware.

Develop with Intel® Software Development Tools.

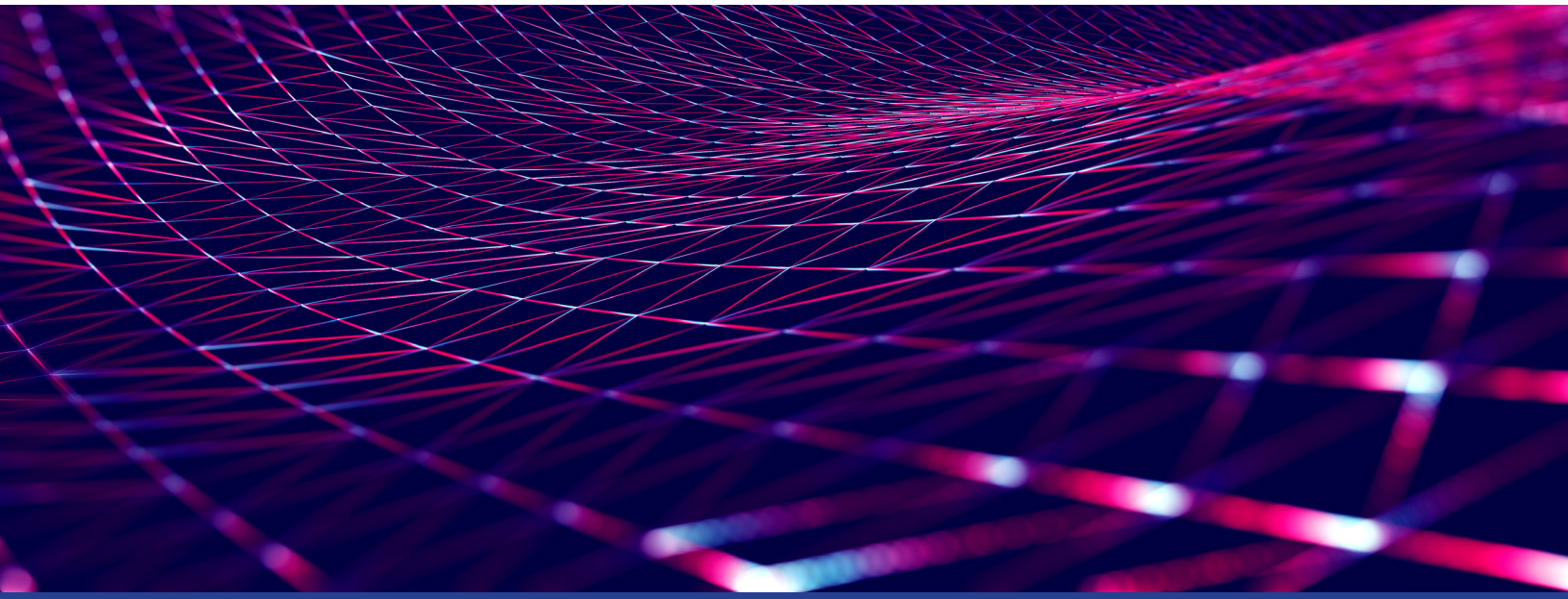
Jump-start your projects without having to download, configure, or install the latest compilers, performance libraries, and tools from Intel.

Use Popular AI Frameworks.

Accelerate your algorithms and applications with Intel® Optimized AI Frameworks that are ready for training and inference.

GET STARTED NOW >

For more complete information about compiler optimizations, see our Optimization Notice at software.intel.com/articles/optimization-notice#opt-en.
Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.
© Intel Corporation



Improving the Performance of XGBoost and LightGBM Inference

Get Up To 36x Faster Inference Using Intel® oneAPI Data Analytics Library

Igor Rukhovich, Machine Learning Intern, Intel Corporation

Gradient boosting on decision trees is one of the most accurate and efficient machine learning algorithms for classification and regression. There are many implementations of gradient boosting, but the most popular are the **XGBoost** and **LightGBM** frameworks. This article will show how to improve the prediction speed of XGBoost or LightGBM models up to 36x with **Intel® oneAPI Data Analytics Library (oneDAL)**.

For more complete information about compiler optimizations, see our Optimization Notice.

Gradient Boosting

Many people use XGBoost and LightGBM gradient boosting to solve various real-world problems, conduct research, and compete in **Kaggle** competitions. Although these frameworks give good performance out of the box, prediction speed can still be improved. Considering prediction is possibly the most important stage of the machine learning workflow, performance improvements can be quite beneficial.

A previous article showed that oneDAL performs gradient boosting inference several times faster than its competitors: **Fast Gradient Boosting Tree Inference**. This performance benefit is now available in XGBoost and LightGBM.

Model Converters

All gradient boosting implementations perform similar operations, and therefore have similar data storage. In theory, this facilitates the conversion of trained models from one machine learning framework to another. Model converters in oneDAL are designed to help you transfer a trained model from XGBoost or LightGBM to oneDAL with just a single line of code. Model converters from other frameworks will soon be available.

The following examples show how to convert XGBoost and LightGBM models to oneDAL. First, get the latest version of **daal4py** for Python 3.6 and higher:

```
conda install -c conda-forge daal4py'>=2020.3'
```

Convert an XGBoost model to oneDAL:

```
# Train an XGBoost model
import xgboost as xgb
clf = xgb.XGBClassifier(**params)
xgb_model = clf.fit(X_train, y_train)

# Convert the XGBoost model to a oneDAL model
import daal4py as d4p
daal_model = d4p.get_gbt_model_from_xgboost(xgb_model.get_booster())

# Make a faster prediction with oneDAL
daal_prediction = d4p.gbt_classification_prediction(nClasses=n_classes)
.compute(X_test, daal_model).prediction
```

For more complete information about compiler optimizations, see our Optimization Notice.

Convert a LightGBM model to oneDAL:

```
# Train a LightGBM model
import lightgbm as lgb
lgb_model = lgb.train(params, lgb.Dataset(X_train, y_train))

# Convert the LightGBM model to a oneDAL model
import daal4py as d4p
daal_model = d4p.get_gbt_model_from_lightgbm(lgb_model)

# Make a faster prediction with oneDAL
daal_prediction = d4p.gbt_regression_prediction().compute(X_test,
daal_model).prediction
```

Note that there is temporary limitation on the use of missing values (`NaN`) during training and prediction. Inference quality might be lower if the data has missing values.

The following example shows how to save and load a model from oneDAL:

```
# Model from XGBoost
daal_model = d4p.get_gbt_model_from_xgboost(xgb_model)

import pickle

# Save model to a file
with open('model.pkl','wb') as out:
    pickle.dump(daal_model, out)

# Load model from a file
with open('model.pkl','rb') as inp:
    model = pickle.load(inp)

# Make predictions
daal_prediction =
d4p.gbt_regression_prediction().compute(X_test, model)
```

By default, oneDAL only returns labels for predicted elements. If you need the probabilities as well, you must explicitly ask for them:

For more complete information about compiler optimizations, see our Optimization Notice.

Sign up for future issues

```
# List all results that you need by placing '|' between them:
predict_algo = d4p.gbt_classification_prediction(nClasses=n_classes,
    resultsToEvaluate="computeClassLabels|computeClassProbabilities")
daal_prediction = predict_algo.compute(X_test, model)

# Get probabilities:
probabilities = daal_prediction.probabilities

# Get labels:
labels = daal_prediction.prediction
```

Performance Comparison

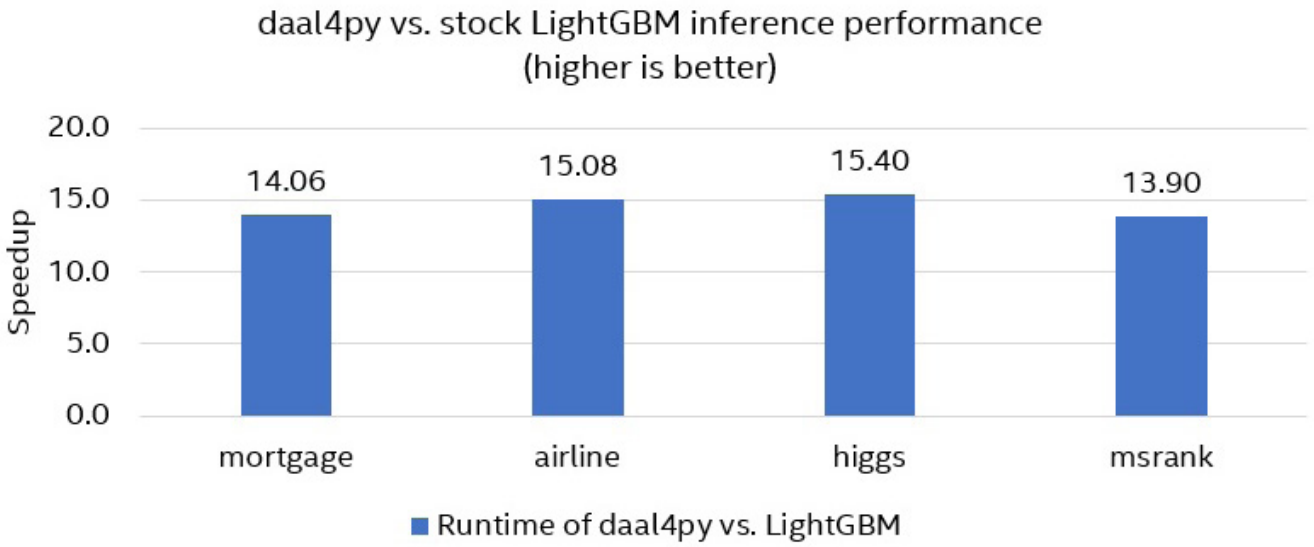
The performance advantage of oneDAL over XGBoost and LightGBM is demonstrated using the following off-the-shelf datasets:

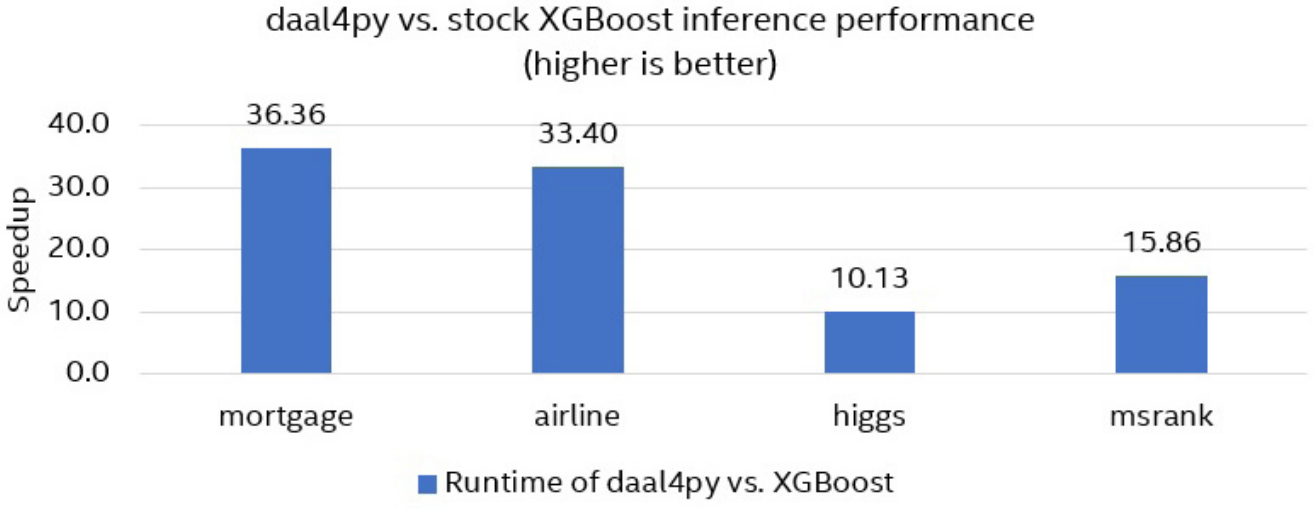
- **Mortgage** (45 features, ~9M observations)
- **Airline** (691 features, one-hot encoding, ~1M observations)
- **Higgs** (28 features, 1M observations)
- **MSRank** (136 features, 3M observations)

The models were trained in XGBoost and LightGBM, then converted to daal4py. To compare performance of stock XGBoost and LightGBM with daal4py acceleration, the prediction times for both original and converted models were measured. **Figure 1** shows that daal4py is up to 36x faster than XGBoost (24x faster on average) and up to 15.5x faster than LightGBM (14.5x faster on average). Note that prediction quality remains the same (as measured by mean squared error for regression and accuracy and logistic loss for classification).

oneDAL uses the **Intel® Advanced Vector Extensions 512 (Intel® AVX-512)** instruction set to maximize gradient boosting performance on **Intel® Xeon® processors**. The most commonly-used inference operations, such as comparison and random memory access, can be effectively implemented using the `vpgatherd{d,q} and vcmpp{s,d}` instructions in AVX-512. Performance also depends on storage efficiency and memory bandwidth. For tree structures, oneDAL uses smart locking of data in memory to achieve temporary cache localization (i.e., the state when a subset of trees and a block of observations are stored in L1 data cache) so that the majority of memory accesses are satisfied immediately at the L1 level with the highest memory bandwidth.

For more complete information about compiler optimizations, see our Optimization Notice.





1 **Comparing daal4py inference performance to XGBoost (top) and LightGBM (bottom). Hardware and software details are below.**

For more complete information about compiler optimizations, see our Optimization Notice.

Final Thoughts

Many applications use XGBoost and LightGBM for gradient boosting, so model converters provide an easy way to accelerate inference using oneDAL. They allow XGBoost and LightGBM users to:

- **Use their existing model training code** without changes.
- **Do inference up to 36x faster** with minimal code changes and no loss of quality.

Hardware and Software Configuration

Intel® Xeon® Platinum 8275CL (2nd generation Intel Xeon Scalable processors): 2 sockets, 24 cores per socket, HT:on, Turbo:on. OS: Ubuntu 18.04.4 LTS (Bionic Beaver), total memory of 192 GB (12 slots/16 GB/2933 MHz). Software: XGBoost 1.2.1, LightGBM 3.0.0, daal4py version 2020 update 3, Python 3.7.9, numpy 1.19.2, pandas 1.1.3, and scikit-learn 0.23.2. Training Parameters: **XGBoost** and **LightGBM**.

For more complete information about compiler optimizations, see our Optimization Notice.

intel® software

THE PARALLEL UNIVERSE

Intel technologies may require enabled hardware, software or service activation. Learn more at intel.com or from the OEM or retailer.

Your costs and results may vary.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804. https://software.intel.com/en-us/articles/optimization-notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Printed in USA 1220/SS Please Recycle