December 11, 2021
Modified (February 6, 2022)

Homogeneous and Heterogeneous Implementations of a tridiagonal solver on Intel® Xeon® E-2176G with oneMKL getrs() routine

By

Oluwatosin Esther Odubanjo

Summary
This project presents homogeneous and heterogeneous implementations of a tridiagonal solver. These solvers have been developed within the Intel® oneAPI framework and have been implemented with API-based programming using oneMKL getrs() routine to exploit the Intel® Xeon® E-2176G, a heterogeneous multi-core processor available on the Intel® DevCloud development environment and to explore the data and memory management, device management and error handling features offered by DPC++ (an integration of C++ and SYCL).

1.1. Characteristics of Development Environment and Processor used

Development Environment
Environment: Intel DevCloud **Accessed from:** HP-15-R029WM, Intel® Pentium® CPU N3540 @2.16GHz, 4 Core(s), 4 Logical Processor(s) **File Storage:** 220GB **RAM:** 192GB **Terminal Interface:** Linux
Characteristics of Processor
Processor: Intel® Xeon® E-2176G **Frequency**: 3.70 GHz (4.70 GHz turbo) **Cache:** 12 MB Intel® Smart Cache **Cores:** 6 (12 Logical) **Threads:** 12 **Max Memory Size:** 128GB **Memory Type:** DDR4-2666 **Max Memory Bandwidth:** 41.6 GB/s **Processor Graphics:** Intel® UHD Graphics P630 [0x3e96] **Frequency:** 0.35 – 1.20 GHz **Execution Units:** 24 (For more information, see: https://www.intel.com/content/www/us/en/products/sku/134860/intel-xeon-e2176gprocessor-12m-cache-up-to-4-70-ghz/specifications.html)

1.2. What is getrs() :
(https://oneapi-src.github.io/oneMKL/domains/lapack/getrs.html#onemkl-lapack-getrs)

The getrs() routine belongs to the oneapi::mkl::lapack namespace, it has a unified shared memory (USM) and Buffer version. It solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides. The getrs() routine supports float, double, std::complex and std::complex precisions and can be executed on Host, CPU and GPU devices.

Since the getrs() routine works with an LU-factored square coefficient matrix the getrf() routine must be called first to compute the LU-factorization of the matrix as A = P*L*U, where P is a permutation matrix, L is the lower triangular matrix with unit diagonal elements (lower trapezoidal if m > n for an m*n matrix) and U is the upper triangular matrix(upper trapezoidal if m < n for an m*n matrix). This routine uses partial pivoting with row interchanges, it has a USM and Buffer version and can be executed on Host, CPU and GPU devices like the getrs() routine.

1.3. What is a Tridiagonal System ?

Tridiagonal linear systems are a special type of banded linear system having non-zero elements on the main-diagonal, sub-diagonal and super-diagonal; they have specific applications in fluid dynamics, computer graphics, modeling of medical problems, finance, and many more.

Sections 1.4 – 1.6 below briefly highlights how the implementations exploits the data and memory management, device management and error handling features offered by DPC++.

1.4. Data and Memory Management

1. **Buffer:** A buffer provides an abstract view of memory that can be accessed on either the host or device

2. **Unified Shared Memory:** The Unified Shared Memory manages memory using the pionter-based approach familiar to C/C++ programmers.

Type of allocation:
Shared allocation. Using malloc_shared, the same data object can be accessed on the host as well as device, that is, data can migrate between host memory and device-local memory.

Method for data movement:
Implicit data movement. In this type of data movement, copy operations do not need to be inserted explicitly to move data between host memory and device-local memory; data movement is handled by runtime mechanisms and lower-level drivers invisible to the programmer.

1.5. Device Management

Queues: They are used to specify what section of code uses a particular device. For the homogeneous implementations, one queue is bined to the CPU or GPU device. For the heterogeneous implementations, two queues are binded to the CPU and GPU devices respectively.

1.6 Error handling

Errors in programs are detected and managed by explicitly handling synchronous and asynchronous errors.

1.7. Implementations

Homogeneous implementation: Implementation is executed on either host or gpu device.
Heterogeneous implementation: Queues are used to specify what section of code uses a particular device.

1. getrs_usm.cpp: Homogeneous implementation based on Unified Shared Memory version of getrs(). This implementation can be executed on either host or gpu device.

2. getrs_buffer.cpp: Homogeneous implementation based on buffer version of getrs implementation.

3. getrs_buffer_het.cpp: Heterogeneous implementation based on buffer version of getrs(). Queues are used to define that:
1. The factorization of the matrix is done on the GPU device;
2. The solution of the matrix is computed on the Host device (in other words, do the getrf() on the GPU and the getrs() on the CPU).

4. getrs_usm_het.cpp: Heterogeneous implementation based on Unified Shared Memory version of getrs(). Queues are used to define that:
1. The factorization of the matrix is done on the GPU device;
2. The solution of the matrix is computed on the Host device (in other words, do the getrf() on the GPU and the getrs() on the CPU).

In this heterogeneous version, memory for the matrix and pivot array are accessed on the gpu device, while that of the right-hand side array is accessed on the host device. Also, the pointer to the scratchpad memory which is used for storing intermediate results as well as the size of the scratchpad memory for getrf() is accessed on the GPU device, while that of getrs() is accessed on the CPU device.

For simplicity, I will call this USM Heterogeneous Version 1.

5. getrs_usm_het2.cpp: Heterogeneous implementation based on Unified Shared Memory version of getrs(). Queues are used to define that:
1. Memory for the matrix, pivot and right-hand side array is accessed on the Host device;
2. The pointer to the scratchpad memory which is used for storing intermediate results as well as the size of the scratchpad memory for both getrf() and getrs() is accessed on the GPU device.
3. Both getrf and getrs are computed on the Host device.

For simplicity, I will call this USM Heterogeneous Version 2.

1.8. Results

Test sizes = 500, 2500, 5000, 10000

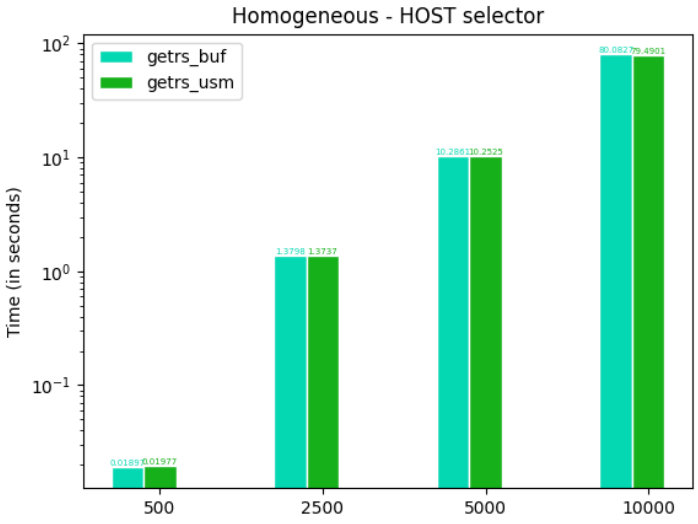
1.8.1. Table of Results:

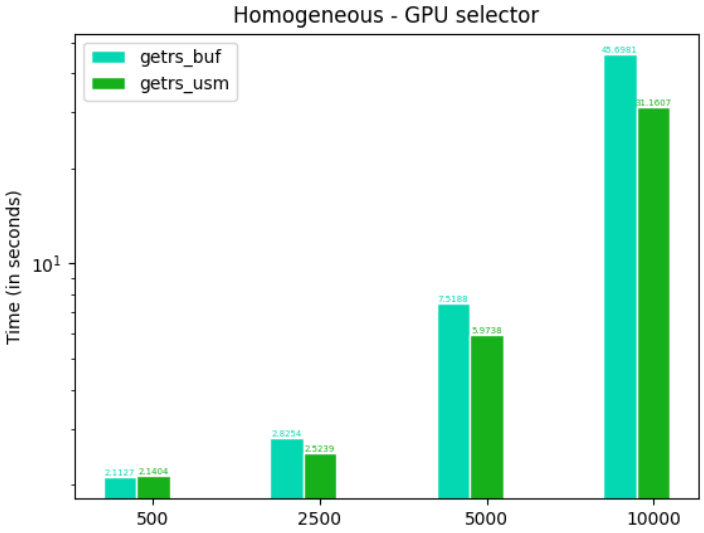
Program	SIZE = 500			
	Homogeneous Implementation			
	T1 (secs)	T2 (secs)	T3 (secs)	Taverage(secs)
	host_selector			
getrs_buffer.cpp	0.01874	0.01895	0.01927	0.01897
getrs_usm.cpp	0.01997	0.01965	0.01969	0.01977
	gpu_selector			
getrs_buffer.cpp	2.1501	2.1154	2.0816	2.1157
getrs_usm.cpp	2.1619	2.1421	2.1171	2.1404
	Heterogeneous Implementation			
getrs_buf_het.cpp	1.8733	1.8853	1.8584	1.8723
getrs_usm_het.cpp	1.9637	1.9522	1.8992	1.9384
getrs_usm_het2.cpp	0.02156	0.01964	0.01988	0.02036
	SIZE = 2500			
	Homogeneous Implementation			
	T1 (secs)	T2 (secs)	T3 (secs)	Taverage(secs)
	host_selector			
getrs_buffer.cpp	1.3819	1.3750	1.3826	1.3798
getrs_usm.cpp	1.3697	1.3782	1.3732	1.3737
	gpu_selector			
getrs_buffer.cpp	2.8690	2.8362	2.7710	2.8254
getrs_usm.cpp	2.4813	2.5946	2.4959	2.5239
	Heterogeneous Implementation			
getrs_buf_het.cpp	3.1995	3.1933	3.1895	3.1941
getrs_usm_het.cpp	3.0826	3.0559	3.0807	3.0731
getrs_usm_het2.cpp	1.3688	1.3693	1.3694	1.3692
	SIZE = 5000			
	Homogeneous Implementation			
	T1 (secs)	T2 (secs)	T3 (secs)	Taverage(secs)
	host_selector			

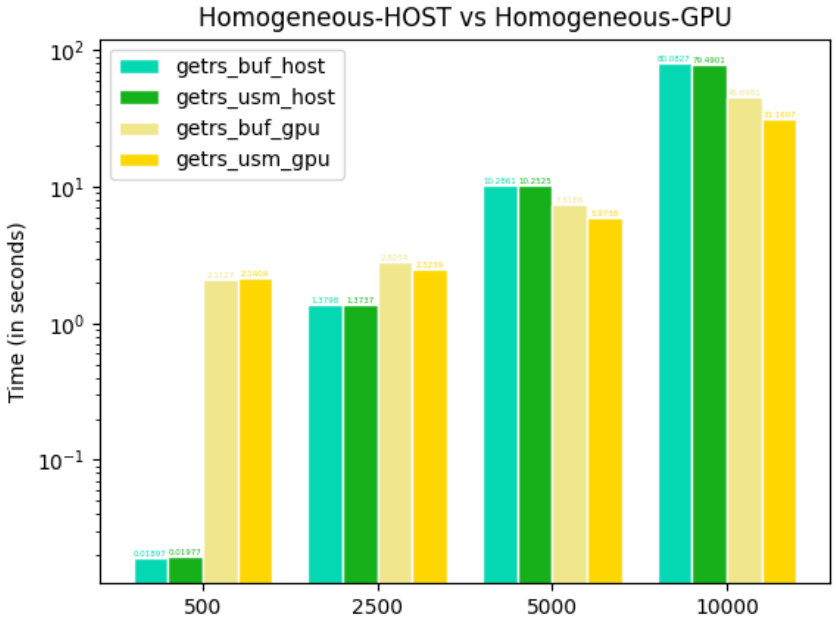
getrs_buffer.cpp	10.2932	10.2780	10.2871	10.2861
getrs_usm.cpp	10.2590	10.2271	10.2713	10.2525
	gpu_selector			
getrs_buffer.cpp	7.4965	7.5154	7.5446	7.5188
getrs_usm.cpp	5.9490	5.9492	6.0233	5.9738
	Heterogeneous Implementation			
getrs_buf_het.cpp	12.2506	12.2845	12.2695	12.2682
getrs_usm_het.cpp	10.8921	10.9011	10.9011	10.8887
getrs_usm_het2.cpp	10.2631	10.2446	10.2427	10.2501
	SIZE = 10000			
	Homogeneous Implementation			
	T1 (secs)	T2 (secs)	T3 (secs)	Taverage(secs)
	host_selector			
getrs_buffer.cpp	80.0665	80.0981	80.0834	80.0827
getrs_usm.cpp	79.4300	79.5188	79.5214	79.4901
	gpu_selector			
getrs_buffer.cpp	45.6599	45.6974	45.7371	45.6981
getrs_usm.cpp	31.1704	31.1466	31.1651	31.1607
	Heterogeneous Implementation			
getrs_buf_het.cpp	83.5490	83.5449	83.3031	83.4657
getrs_usm_het.cpp	69.7037	69.6627	69.6609	69.6758
getrs_usm_het2.cpp	79.6955	79.5260	79.6392	79.6202

1.8.2. Graphical representation of results:

1.8.2.1. Homogeneous Implementations:







Comments:

The figures above shows:

(1) the results for Homogeneous getrs USM and BUFFER implementations executed on host device.

(2) the results for Homogeneous getrs USM and BUFFER implementations executed on gpu device.

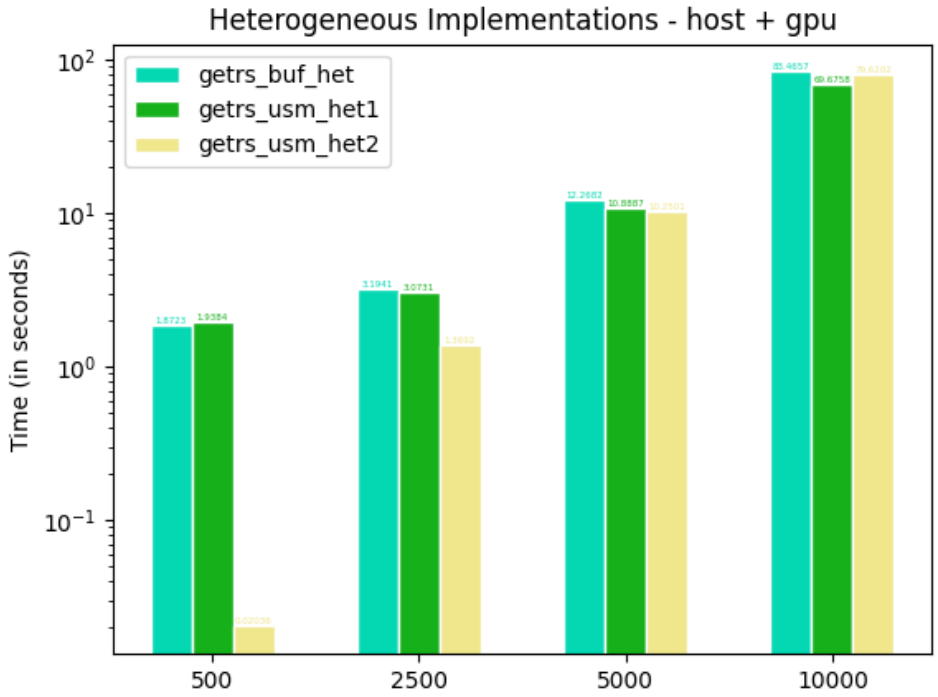
(3) comparison of (1) and (2)

What can we see?

1. On the host, the getrs USM version executes faster (slightly, though) than the getrs buffer version for the test sizes greater than 500. This same behaviour (but not slightly faster, it is faster) is observed for both versions on the gpu device.

2. Both the getrs USM and buffer versions executes faster on the gpu device than on the host device for test sizes greater that 2500.

1.8.2.2. Heterogeneous Implementations:



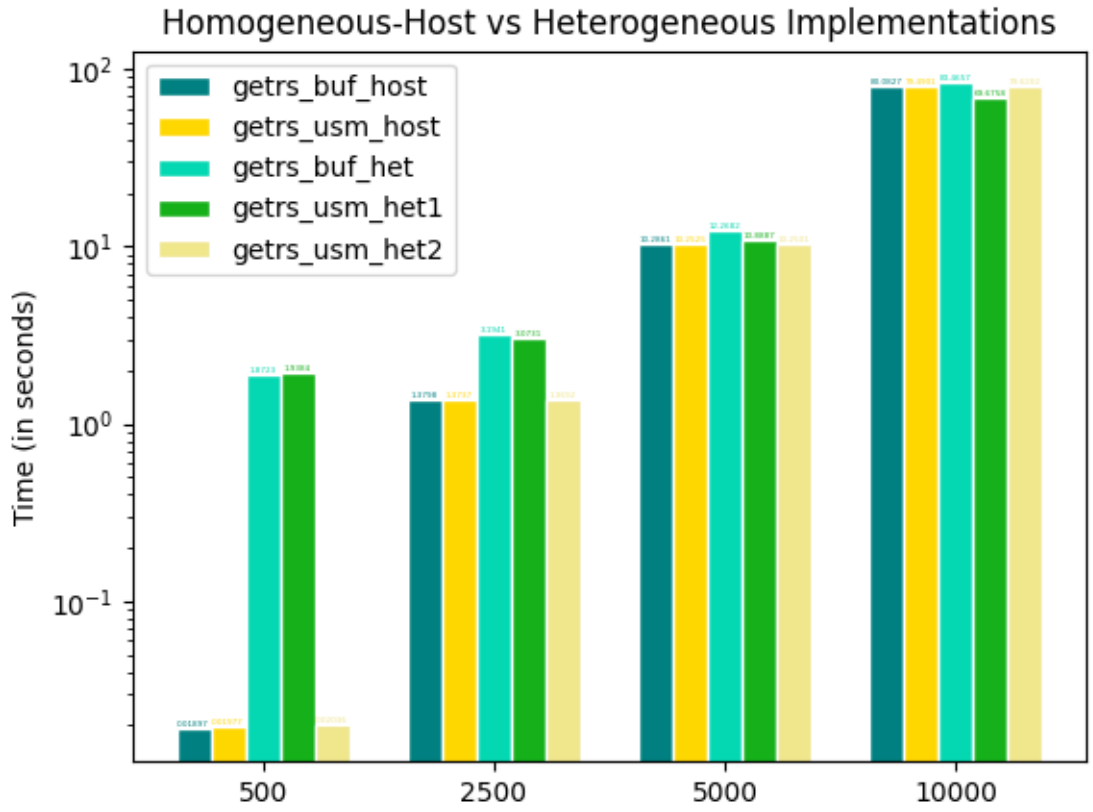
Comment:

The figure above shows the results for the getrs buffer and USM heterogeneous implementations.

What can we see?

The getrs USM heterogeneous version 2 executes faster for the test sizes less than 10000. At test size 10000, the getrs USM heterogeneous version 1 is the best performing heterogeneous implementation.

1.8.2.3. Homogeneous-host Implementations versus Heterogeneous Implementation

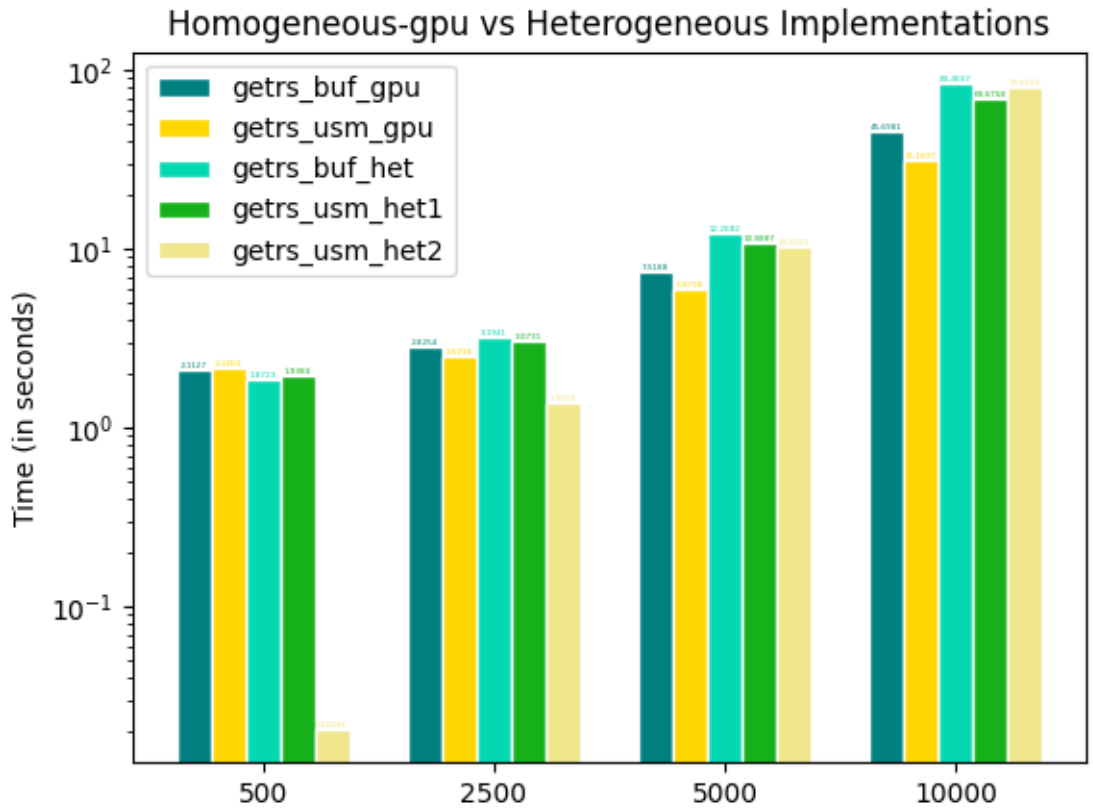


Comment:

The figure above shows an execution time comparison for the getrs homogeneous implementations (USM and BUFFER) executed on the host device and heterogeneous implementations executed on both the host and gpu devices.

For test size 500, the getrs buffer implementation executed on the host, executes faster than the getrs USM heterogeneous implementations. However, this is not so for test sizes 2500 – 10000; the getrs USM heterogeneous version 2 executes faster for test sizes 2500 and 5000 while the getrs USM heterogeneous version 1 executes faster for test size 10000.

1.8.2.4. Homogeneous-gpu Implementations versus Heterogeneous Implementation



Comment:

The figure above shows an execution time comparison for the getrs homogeneous implementations (USM and BUFFER) executed on the gpu device and heterogeneous implementations executed on both the host and gpu devices.

For test sizes 500 and 2500, the getrs USM heterogeneous version 2 executes faster than the homogeneous versions executed on the gpu; however, for test size 5000 and 10000, this is not so, the getrs USM homogeneous implementation executed on the gpu, executes faster.

1.8.2.5. Choosing the best implementation based on execution time

The best implementation is relative to the choice of device and data size. The table below shows the best performing programs for the homogeneous and heterogeneous implementations.

Homogeneous Implementation	
SIZE	**Program**
500	getrs_buffer.cpp (host_selector)
2500	getrs_usm.cpp (host_selector)
5000	getrs_usm.cpp (gpu_selector)

10000	getrs_usm.cpp (gpu_selector)
Heterogeneous Implementation	
SIZE	**Program**
500	getrs_usm_het2.cpp
2500	getrs_usm_het2.cpp
5000	getrs_usm_het2.cpp
10000	getrs_usm_het.cpp
Overall best Implementation	
SIZE	
500	getrs_buffer.cpp (host_selector)
2500	getrs_usm_het2.cpp
5000	getrs_usm.cpp (gpu_selector)
10000	getrs_usm.cpp (gpu_selector)

Final Comments:

1. In solving the tridiagonal system, the buffer version of oneMKL getrs (host_selector) excels for a small test size.

2. In solving the tridiagonal system, the USM version of oneMKL getrs excels for large test sizes.

3. For large test sizes, the gpu execution is faster compared to cpu and heterogeneous executions. This is a testimony to the fact that gpus have access to more processors than a cpu, they are well adapted for problems that can be expressed as a data parallel computation. Furthermore, it is important to note that, the P360 gpu is embedded in the E-2176G Processor; it is a co-processor with its host, in this case its host is a cpu.

4. For large test sizes, the next overall best performing implementation after the getrs homogeneous (gpu_selector) implementation is the getrs USM heterogeneous version 2 for test size (500 – 5000) and getrs USM heterogeneous version 1 for test size 10000. This shows the effectiveness of utilizing all devices available on a heterogeneous processor.

Read about DPC++:

J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogenous Systems using C+ and SYCL.* Berkely, CA: Apress, 2021. [Online]. Available: https://doi.org/10.1007/978-1-4842-5574-2