IRIS-BLAS: Towards a Performance Portable and Heterogeneous BLAS Library

1st Narasinga Rao Miniskar
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, USA
miniskarnr@ornl.gov

2nd Mohammad Alaul Haque Monil
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, USA
monilm@ornl.gov

3rd Pedro Valero-Lara
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, USA
valerolarap@ornl.gov

4th Frank Liu
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, USA
liufy@ornl.gov

5th Jeffrey S. Vetter
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, USA
vetter@ornl.gov

***Abstract*—This paper presents IRIS-BLAS, a novel heterogeneous and performance portable BLAS library.IRIS-BLAS is built on top of the IRIS runtime and multiple vendor and open-source BLAS libraries. It can transparently use all the architectures/devices available in a heterogeneous system, using the appropriate BLAS library based on the task mapping at run time. Thus, IRIS-BLAS is portable across a broad spectrum of architectures and BLAS libraries, alleviating the worry of application developers about modifying the application source code. Even though the emphasis is on portability, IRIS-BLAS provides competitive or even better performance than other state-of-the-art references. Moreover, IRIS-BLAS offers new features such as efficiently using extremely heterogeneous systems composed of multiple GPUs from different hardware vendors.**

***Index Terms*—Performance Portable, Heterogeneity, IRIS, BLAS, Tasking.**

I. Introduction & Related Work

With the explosion of heterogeneity in current and future computer systems, HPC math libraries are facing important challenges [1]. Manually attempting to coordinate and schedule data placement/computation, which types of processors to select for certain calculations, and when computations will occur is becoming an intractable problem due to the growing complexity, diversity, and scale of HPC systems. Historically, BLAS (Basic Linear Algebra Subprograms) vendors [2]–[4] and open-source [5]–[7] libraries are only focused on one architecture and/or programming model. Although we can find some examples [8], [9] using both, CPU and GPU, for some math kernels, they require programmers to decide which device to use for each component of the problem or/and computationally expensive pre-processing to identify which device to use.

Task-based dynamic runtimes are positioned as part of the solution to the challenges aforementioned [1]. Novel developments are needed in software abstractions to increase application portability using fully dynamic runtime systems to schedule and control highly varied resources. However, existing approaches rely mostly on static mapping/scheduling where the programmer must decide which device to use for each task, such as OpenMP tasking [10] or others [11]. Fortunately, we can find two dynamic runtime systems; StarPU [12] and IRIS [13]. The later provides a higher capacity and support for a greater variety of devices and programming models [1]. In this work, we compare the performance of both systems.

The main motivation and objective of this work is to provide a performance portable and heterogeneous BLAS library that is able to use all kinds of architectures hosted by any computer node, using a fully dynamic runtime system, and so, pursue the maximum performance that we can get independently of the complexity, diversity, and scale of the computer system to use.

II. Background: IRIS

IRIS [13] is a programming system for extremely heterogeneous architectures. Figure 1 illustrates its architecture. IRIS enables application developers to write portable applications across diverse heterogeneous programming platforms including CUDA, HIP, Level Zero, OpenCL, and OpenMP. It orchestrates

[1]https://iris-programming.readthedocs.io/en/latest/

multiple programming platforms in a system into a single execution/programming environment by providing portable tasks and shared virtual device memory.

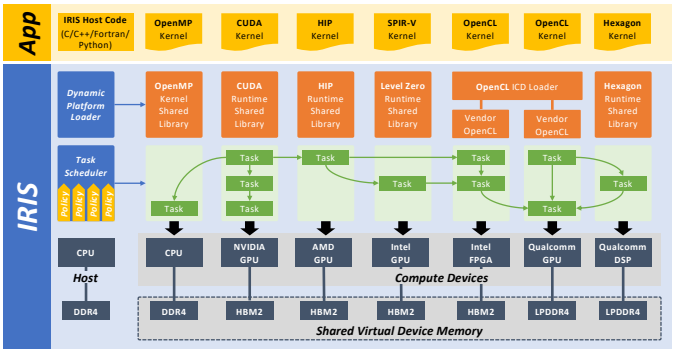

Fig. 1: The IRIS architecture

IRIS provides a task-based programming model. A task in IRIS is a scheduling unit. A task runs on a single device, and it is portable across any compute devices in the system, including accelerators, such as AMD and NVIDIA GPUs, FPGAs, among others. A task contains zero or more commands. There are four types of commands: host-to-device memory copy command, device-to-host memory copy command, kernel launch command, and host command. A task can have a dependency on other tasks. When a task depends on other tasks, it cannot start until the prerequisite tasks complete. Therefore, writing an IRIS application means to build directed acyclic graphs of tasks. Each task has a target device selection policy when it is submitted. The policy is specified by programmers, and it can be a device number, type (CPU, GPU, FPGA, or DSP), or builtin policies such as greedy, random, locality-aware, and profile, provided by IRIS.

To achieve application portability and flexible task scheduling with effective data orchestration, IRIS provides shared virtual device memory across multiple, disjoint physical device memories. IRIS automatically transfers data across multiple devices to keep memory consistency across tasks. Therefore, all compute devices can share memory objects in the shared virtual device memory, and they can see the same content in the memory objects.

Table I shows a feature comparison between IRIS and StarPU [12] runtime. IRIS and StarPU share some striking similarities; for example, tasks and kernels in the IRIS are conceptually equivalent to the tasks and codelets in the StarPU. Even though StarPU is one of the pioneers in the heterogeneous runtime domain, IRIS supports a broader range of accelerators, programming models, and BLAS libraries. However, StarPU being more mature supports some extra features that IRIS does not yet, such as CUDA streaming.

III. IRIS-BLAS

The overall IRIS-BLAS software stack is shown in Figure 2. It is supported with OpenBLAS [7], Intel MKL [2], NVIDIA cuBLAS [3], CLBlast [14] (Adreno, NVIDIA and AMD GPUs), AMD hipBLAS [15] vendor specific and open-source BLAS libraries.

TABLE I: IRIS and StarPU comparison

Feature	IRIS	StarPU
Architectures	CPU, GPUs (Nvidia & AMD), FPGAs (Intel & Xilinx), Qualcomm SoCs	CPU, GPUs, (Nvidia & AMD), Xilinx FPGAs
Programming Models	OpenMP, CUDA, HIP, OpenCL, IntelCL, XilinxCL, HexagonC	OpenMP, CUDA, HIP, OpenCL
BLAS Libraries	OpenBLAS, Intel MKL, ATLAS, cuBLAS, hipBLAS rocBLAS, CLBLAST	OpenBLAS, Intel MKL, cuBLAS, ATLAS, GOTO
CUDA Streams	Not supported	Supported

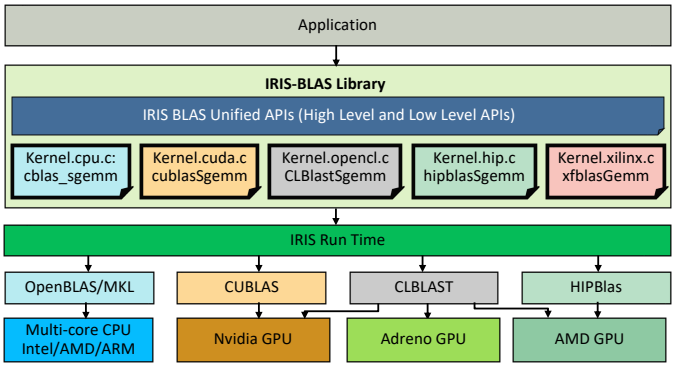

Fig. 2: IRIS-BLAS Software Stack

IRIS-BLAS is able to decide which device to use at run-time. However, the BLAS library to use on each device must be decided at compile time. For example, developers can decide which CPU BLAS library to use (OpenBLAS, Intel MKL, cBLAS) at compile time. IRIS-BLAS supported libraries and architectures is shown in Figure 2.

A. IRIS-BLAS APIs

IRIS-BLAS provides developers two different APIs for each BLAS operation:

1) High-level API (AI): High-level APIs are for the developers to call the IRIS-BLAS API in the place of traditional OpenBLAS/MKL/cuBLAS/hipBLAS API, just changing the name of the functions. It takes the same host address space pointers for both input and output parameters (matrices and arrays) similar to traditional BLAS API. IRIS-BLAS API implementation takes care of creating IRIS task, IRIS memory objects and IRIS kernel. It also takes care of introducing host to device and device to host data transfer commands in the IRIS task. It simplifies the application programming to migrate from compute specific BLAS operation to heterogeneous BLAS operation with a simple name change of function.
2) Low-level API : This API is for advanced programming developers whom want to share the IRIS memory objects across different IRIS tasks to save the data transfers. It provides developers full control on memory management

and task dependencies, so that the library can better inter-operate with other IRIS tasks or applications/libraries. This API implementation handles IRIS task kernel creation by setting the appropriate parameters of the kernel. When compared to high-level API, it takes IRIS task parameter as an input to insert the task kernel.

For example, as shown in Figure 3, we provide the two IRIS-BLAS APIs for *SGEMM* BLAS Level-3 routine. The only difference between the two APIs is the function arguments. The high-level API *iris_core_sgemm* requires only the host-memory pointers. This is a blocking API, which takes care of task creation/submission, tuning, memory management and data transfers. This API also requires one additional parameter; *target_dev* in Figure 3. This argument can be used to specify which device to use (CPU, GPU, FPGA, etc.) or we can leave the decision about which device to use to IRIS (using *iris_all*). The other API, *iris_task_sgemm* is similar to the previous one in the implementation, but provides more control to application programmer on memory management, task dependencies, as well as on task submission either directly or using task graph data structure. In the case of the semi-control API, users have to provide only IRIS memory handlers/objects in the place of host pointers. Data transfers will be taken care by IRIS based on the access of IRIS memory objects and task kernel execution on the target device. This API doesn't take *target_dev* as an argument. Low-level APIs are necessary for programmer to share the IRIS memory handlers between different tasks, and so IRIS can optimize the data transfers between dependent tasks based on the device mapping of tasks.

```
int iris_core_sgemm( int target_dev, IRISBlasType major,
            IRISBlasType a_trans, IRISBlasType b_trans,
            int M, int N, int K, float alpha,
            float *A, int lda,
            float *B, int ldb, float beta,
            float *C, int ldc ) {
  IRIS_SINGLE_TASK( task0, "iris_sgemm_kernel",
    target_dev, 1,
    NULL_OFFSET, GWS(M), NULL_LWS,
    PARAM(IRISBlasType, major),
    PARAM(IRISBlasType, a_trans), ...
    IN_TASK(A, float*, float, A, sizeof(float)*M*N),
    PARAM(int, lda), ...
    IN_OUT_TASK(C, float*, float, C, sizeof(float)*M*K),
    PARAM(int, ldc) );
  return IRIS_SUCCESS;
}
int iris_task_sgemm( iris_task task0, IRISBlasType major,
            IRISBlasType a_trans, IRISBlasType b_trans,
            int M, int N, int K, float alpha,
            iris_mem IRIS_VAR(A), int lda,
            iris_mem IRIS_VAR(B), int ldb, float beta,
            iris_mem IRIS_VAR(C), int ldc ) {
  IRIS_TASK_NO_DT( task0, "iris_sgemm_kernel", 1,
    NULL_OFFSET, GWS(M), NULL_LWS,
    PARAM(IRISBlasType, major),
    PARAM(IRISBlasType, a_trans), ...
    IN_TASK(A, float*, float, A, sizeof(float)*M*N),
    PARAM(int, lda), ...
    IN_OUT_TASK(C, float*, float, C, sizeof(float)*M*K),
    PARAM(int, ldc) );
    return IRIS_SUCCESS;
}
```

Fig. 3: SGEMM IRIS-BLAS high-level (top) and low-level (bottom) APIs.

Each API is implemented by using a set of IRIS macros, which are used to easily convert any function call to IRIS tasks. These macros (*IRIS_SINGLE_TASK* and *IRIS_TASK_NO_DT* in Figure 3) are also used in the process of creating the host wrapper codes (boiler plate code for IRIS task). As we see later, these wrapper codes are necessary for the interoperability between IRIS-BLAS API, kernel codes, vendor/open-source BLAS libraries, and IRIS runtime. Other macros (*IN_TASK*, *OUT_TASK*, and *IN_OUT_TASK*) are used to indicate the memory parameters whether they are input or output or both input/output, which needs to be fetched using IRIS commands H2D (Host to Device), D2H (Device to Host) or both. Finally, the *PARAM* macro are used for scalar parameters.

B. Core kernels

For example, the core CUDA kernel of SGEMM BLAS routine is shown in Figure 4. IRIS-BLAS only requires core kernel codes and the three application programming interface codes to support one BLAS routine. This approach allows us to create a truly heterogeneous BLAS library with support of different vendor-specific and/or open-source BLAS libraries. IRIS enables the library and compute unit to be selected at run-time based on the task scheduler and device availability.

```
int iris_sgemm_kernel( int devno,
  IRISBlasType major,
  IRISBlasType a_trans, IRISBlasType b_trans,
  int32_t M, int32_t N, int32_t K,
  float alpha, CUdeviceptr A, int32_t lda,
               CUdeviceptr B, int32_t ldb,
  float beta, CUdeviceptr C, int32_t ldc ) {
    cublasStatus_t status =
    cublasSgemm( GetCUBlasHandle(devno),
                 GetCUBlasMap(b_trans),
                 ↪ GetCUBlasMap(a_trans),
                 N, M, K,
                 &alpha, B, ldb,
                         A, lda,
                  &beta, C, ldc );
  if (status != cudaSuccess) return IRIS_ERROR;
  return IRIS_SUCCESS;
}
```

Fig. 4: cuBLAS SGEMM core kernel.

```
int iris_sgemm_kernel( int devno,
  IRISBlasType major,
  IRISBlasType a_trans, IRISBlasType b_trans,
  int32_t M, int32_t N, int32_t K,
  float alpha, float *A, int32_t lda,
               float *B, int32_t ldb,
  float beta,  float *C, int32_t ldc ) {
  cblas_sgemm(GetOpenBlasMap(major),
        GetOpenBlasMap(a_trans), GetOpenBlasMap(b_trans),
        M, N, K, alpha, A, lda, B, ldb, beta, C, ldc);
  return IRIS_SUCCESS;
}
```

Fig. 5: OpenBLAS/MKL SGEMM core kernel.

C. A transparent and highly productivity BLAS framework

The current IRIS framework have native run-time system support for CUDA, HIP and OpenCL as shown in Figure 6.a.

It enables the programmer to provide kernels written directly in CUDA/HIP/OpenCL and doesn't require any wrapper. However, the OpenMP kernels require wrapper codes to gather the kernel parameters and pass them to OpenMP kernel function during the execution. Note that StarPU framework requires wrappers for CUDA, HIP, OpenCL and OpenMP codes.

Unlike the current IRIS framework, the use of vendor specific and open-source libraries requires wrapper codes. IRIS runtime is extended to call these wrapper codes as shown in Figure 6.b. These wrappers are tedious to write manually. Hence, we developed a wrapper code generator tool written in Python, which can parse the IRIS task specification using macros and generates the wrapper code during the compilation (as shown in Figure 6.c). This enables the programmers to use both IRIS-BLAS and native kernels written in CUDA/HIP/OpenCL/OpenMP/etc., providing a fully inter-operable programming environment. Also, IRIS-BLAS is scalable to any new BLAS library by writing only new kernel cores (as shown in Figure 5).

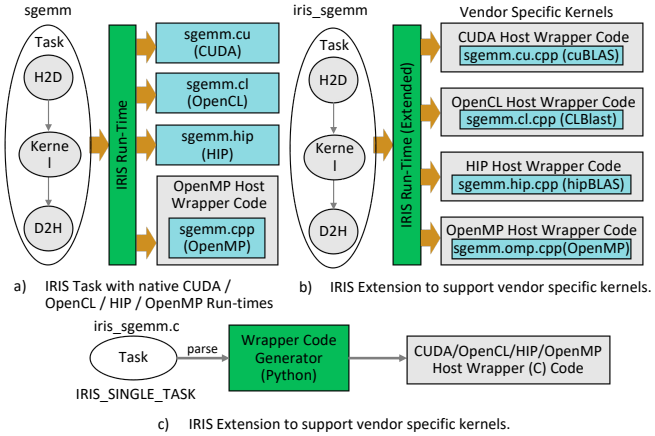

Fig. 6: IRIS Task to Run-time framework with extensions

D. Application Interface

```
#include "irisblas.h"
int call_iris_blas_sgemm(int N) {
  float *A = (float*) malloc(N*N*N*sizeof(float));
  float *B = (float*) malloc(N*N*N*sizeof(float));
  float *C = (float*) malloc(N*N*N*sizeof(float));
  init_matrix(A, N); init_matrix(B, N);
  iris_core_sgemm(iris_any, IRISBLAS_ROW_MAJOR,
        IRISBLAS_NO_TRANS, IRISBLAS_NO_TRANS,
        N, N, N, 1.0, A, N, B, N, 1.0, C, N);
  print_matrix(A, N);
  print_matrix(B, N);
  print_matrix(C, N);
  free(A); free(B); free(C);
  return IRIS_SUCCESS;
}
```

Fig. 7: Example usage of IRIS BLAS Sgemm in application code

An example about how to use IRIS-BLAS SGEMM high level API (*iris_core_sgemm*) for square matrices of size ($N * N$) is shown in Figure 7. It uses the host memory addresses for A, B, and C matrices. IRIS-BLAS is also supported with Python wrappers. An example of using the python IRIS-BLAS SGEMM API is shown in Figure 8. It uses numpy arrays for inputs and outputs.

```
import irisblas
def call_iris_blas_sgemm():
    x = np.array([[1.0, 2.0], [3.0, 4.0]], np.float)
    y = np.array([[1.0, 2.0], [3.0, 4.0]], np.float)
    z = np.array([[0.0, 0.0], [0.0, 0.0]], np.float)
    irisblas = IRISBLAS()
    irisblas.call(irisblas.iris_core_sgemm,
            iris.iris_gpu,
            irisblas.IRIS_BLAS_ROW_MAJOR,
            irisblas.IRIS_BLAS_NO_TRANS,
            irisblas.IRIS_BLAS_NO_TRANS,
            x[0].size, y[0].size, y[1].size,
            np.float(1.0), x, x[0].size,
                           y, y[0].size,
            np.float(1.0), z, z[0].size)
    print('x=',x)
    print('y=',y)
    print('z=',z)
    irisblas.finalize()
```

Fig. 8: Python application with IRIS BLAS Sgemm call

E. Tiled GEMM Implementation

We have implemented a heterogeneous tiled GEMM code in IRIS-BLAS. This code is able to effectively exploit all the heterogeneous resources in the system and distribute the workload/tasks using IRIS run-time scheduler. The use of the task-based programming model implemented in IRIS enables that the decision about which device to use is left to IRIS run-time scheduler. This decisions are taken based on the scheduling algorithm, resource availability and constraints. Similar to the tiled GEMM operation in StarPU [16], the A and B matrices are split into smaller 2-D tiles and the matrix multiplication is carried out on these tiles. Each matrix tile computation is considered as a IRIS task with H2D, Kernel and D2H commands. The accumulation of these tiled matrix multiplications require a *column-sum* operation to accumulate the results. We implemented this kernel using native languages such as CUDA, HIP, OpenCL and OpenMP. This effort helps IRIS run-time for a better task/device mapping, and optimize some of the H2D and D2H data transfers based on the device/task mapping carried out. All these is completely transparent to programmers.

IV. Experimental Results

Next, we analyze IRIS-BLAS in terms of performance portability and heterogeneity support.

A. Performance Portability

We have used different CPU (Intel Xeon-Skylake, AMD 249 EPYC-7763 and Qualcomm Snapdragon) and GPU (Nvidia's A100, AMD's MI100, and Qualcomm Snapdragon Adreno GPUs) architectures. IRIS-BLAS is portable for all supported libraries. As graphically illustrated in Figure 9, a single IRIS-BLAS code/application (SGEMM application) can be run on any architecture. Our library offloads any vendor or open-source optimized BLAS kernels to the corresponding target hardware at runtime. In this case, CLBlast can be offloaded to NVIDIA

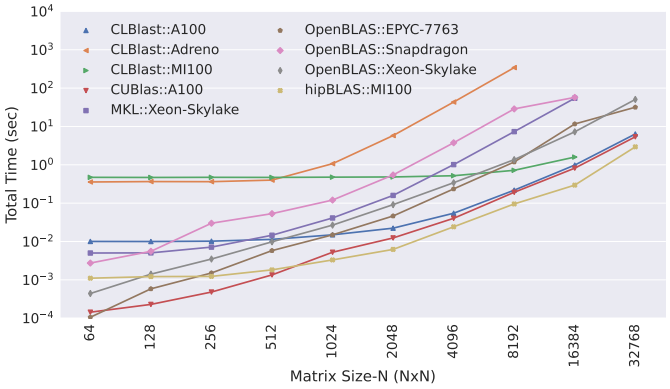

Fig. 9: IRIS-BLAS SGEMM performance on different CPU and GPU architectures.

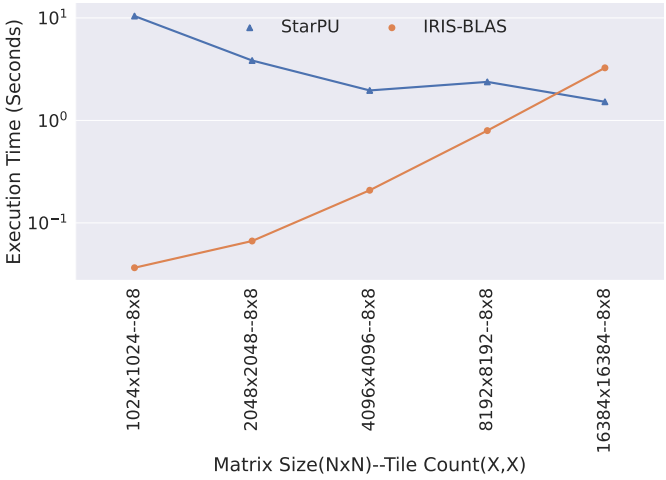

Fig. 10: Comparison of IRIS-BLAS with StarPU-BLAS

A100, AMD MI100 and Qualcomm Adreno GPUs, cuBLAS to NVIDIA A100, MKL to Intel Xeon-Skylake, OpenBLAS to Intel Xeon-Skylake, AMD EPYC-7763 and Qualcomm Snapdragon, and hipBLAS to AMD MI100. Such flexibility in terms of offloading opens the door for advanced scheduling in contemporary and upcoming heterogeneous systems. For this analysis, we have use the high-level SGEMM IRIS-BLAS API. The total time includes the input and output data transfer times along with kernel execution time (high-level API).

Although the objective of IRIS-BLAS is mainly portability, we have measured the performance of IRIS-BLAS for SGEMM operations on different architectures and BLAS libraries. For instance, we can observe that hipBLAS on MI100 is very competitive for big matrix size w.r.t. cuBLAS on A100, OpenBLAS is better than the MKL library on Intel Xeon Skylake, and OpenBLAS on ARM cores achieve better performance than CLBlast on Adreno GPU.

B. StarPU vs IRIS

We have compared the tiled SGEMM implementation in StarPU against our IRIS-BLAS implementation. As in the previous analysis, the IRIS-BLAS implementation uses the high-level SGEMM IRIS-BLAS API. Both codes, StarPU and IRIS-BLAS, use the cuBLAS library and 4 NVIDIA A100 GPUs. We have used different matrix size, with 8×8 number of tiles. As shown (Figure 10), although StarPU has support for CUDA streaming, IRIS-BLAS is able to achieve better or at least competitive performance w.r.t. StarPU implementation, due to the IRIS optimized data transfers, which can avoid some of the data transfers if the data is already in GPU device. As shown the impact of CUDA streams used by StarPU is important on big matrices. This allows to overlap communication with computation, and get better performance than IRIS-BLAS.

C. Support for Extreme-Heterogeneous Systems

To the best of our knowledge, this is the first time that different GPU architectures (4x NVIDIA GPUs and 4x AMD GPUs) and one CPU architecture (1x AMD EPYC 7763 64-core CPU) were used in a parallel and collaborative way (see Figure 11), thanks to the capacity provided by IRIS-BLAS. Note, that StarPU doesn't have this capacity, so we couldn't compare performance against such library. We used as test case, our tiled SGEMM implementation on a matrix of size equal to $16,384 \times 16,384$ with 8×8 count of tiles. We used the IRIS dynamic task scheduling policy *ANY*, which assigns the available resource using a first-come-first-serve basis. Though there are some gaps in the trace illustrated in Figure 11, where some of the computational resources are idle at some point, we can see that all the computational units are being used most of the time.

V. Conclusion & Future Work

We presented some initial results of IRIS-BLAS, a novel, heterogeneous and portable BLAS library, implemented on top of the task-based run-time IRIS. This library support a large number of current HPC architectures, programming models and BLAS libraries. Also, it can inter-operate with other programming models or libraries in a simple manner. IRIS-BLAS is scalable to other architecture and/or BLAS library without much effort. In terms of performance, we are competitive with other reference libraries, such as StarPU, providing even better results. Finally, we were able to use an extreme heterogeneity system in an efficient way.

These initial results and effort open the door to more opportunities towards a better exploitation of current and future heterogeneous systems. Also, we will continue working for a better support on IRIS and IRIS-BLAS, including feature such as streaming tasks. These new supports will not need any change in IRIS-BLAS API, being this one of the advantages of task-based programming model.

Acknowledgments

Notice: This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to

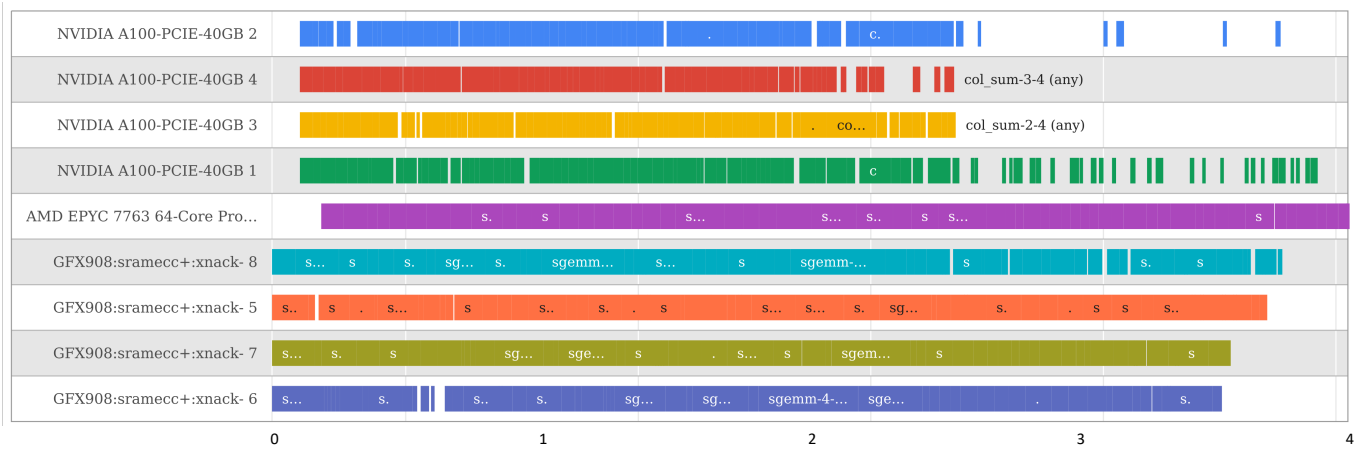

Fig. 11: Trace of a SGEMM IRIS-BLAS code on an extreme heterogeneous platform.

publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] J. Ang, A. A. Chien, S. D. Hammond, A. Hoisie, I. Karlin, S. Pakin, J. Shalf, and J. Vetter, "Reimagining Codesign for Advanced Scientific Computing: Report for the ASCR Workshop on Reimaging Codesign," 2022, [Online; accessed 6-July-2022]. [Online]. Available: https://www.osti.gov/biblio/1822199

[2] Intel, "The Intel Math Kernel Library," 2022, [Online; accessed 6-July-2022]. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html?s=Newest

[3] NVIDIA, "cuBLAS, the CUDA Basic Linear Algebra Subroutine library," 2022, [Online; accessed 6-July-2022]. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[4] AMD, "rocBLAS User Guide," 2022, [Online; accessed 6-July-2022]. [Online]. Available: https://rocblas.readthedocs.io/en/rocm-5.2.0/

[5] J. J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, P. Wu, I. Yamazaki, A. YarKhan, M. Abalenkovs, N. Bagherpour, S. Hammarling, J. Šístek, D. Stevens, M. Zounon, and S. D. Relton, "PLASMA: parallel linear algebra software for multicore using openmp," *ACM Trans. Math. Softw.*, vol. 45, no. 2, pp. 16:1–16:35, 2019. [Online]. Available: https://doi.org/10.1145/3264491

[6] P. Valero-Lara, S. Catalán, X. Martorell, T. Usui, and J. Labarta, "slass: A fully automatic auto-tuned linear algebra library based on openmp extensions implemented in ompss (lass library)," *J. Parallel Distributed Comput.*, vol. 138, pp. 153–171, 2020. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.12.002

[7] M. K. Zhang Xianyi, "OpenBLAS," 2022, [Online; accessed 6-July-2022]. [Online]. Available: https://www.openblas.net/

[8] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.

[9] P. Valero-Lara, A. Pinelli, and M. Prieto-Matías, "Fast finite difference poisson solvers on heterogeneous architectures," *Comput. Phys. Commun.*, vol. 185, no. 4, pp. 1265–1272, 2014. [Online]. Available: https://doi.org/10.1016/j.cpc.2013.12.026

[10] P. Valero-Lara, J. Kim, O. Hernandez, and J. S. Vetter, "Openmp target task: Tasking and target offloading on heterogeneous systems," in *Euro-Par 2021: Parallel Processing Workshops - Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. Chaves, D. B. Heras, A. Ilic, D. Unat, R. M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S. L. Scott, and L. Ricci, Eds., vol. 13098. Springer, 2021, pp. 445–455. [Online]. Available: https://doi.org/10.1007/978-3-031-06156-1_35

[11] O. Korakitis, S. G. D. Gonzalo, N. Guidotti, J. P. Barreto, J. C. Monteiro, and A. J. Peña, "Towards ompss-2 and openacc interoperation," in *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, J. Lee, K. Agrawal, and M. F. Spear, Eds. ACM, 2022, pp. 433–434. [Online]. Available: https://doi.org/10.1145/3503221.3508401

[12] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*, ser. Lecture Notes in Computer Science, H. J. Sips, D. H. J. Epema, and H. Lin, Eds., vol. 5704. Springer, 2009, pp. 863–874. [Online]. Available: https://doi.org/10.1007/978-3-642-03869-3_80

[13] J. Kim, S. Lee, B. Johnston, and J. S. Vetter, "IRIS: A portable runtime system exploiting multiple heterogeneous programming systems," in *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*. IEEE, 2021, pp. 1–8. [Online]. Available: https://doi.org/10.1109/HPEC49654.2021.9622873

[14] C. Nugteren, "Clblast: A tuned opencl blas library," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3204919.3204924

[15] AMD, "hipBLAS, the Basic Linear Algebra Subroutine library," 2022, [Online; accessed 6-July-2022]. [Online]. Available: https://github.com/ROCmSoftwarePlatform/hipBLAS

[16] R. Carratalá-Sáez, M. Faverge, G. Pichon, G. Sylvand, and E. S. Quintana-Ortí, "Tiled algorithms for efficient task-parallel matrix solvers," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 757–766.