Assessing the GPU Offload Threshold of GEMM and GEMV Kernels on Modern Heterogeneous HPC Systems

Finn Wilkinson
High-Performance Computing Group
University of Bristol
Bristol, United Kingdom
fw17231@bristol.ac.uk

Alex Cockrean
High-Performance Computing Group
University of Bristol
Bristol, United Kingdom
no22498@bristol.ac.uk

Wei-Chen Lin
High-Performance Computing Group
University of Bristol
Bristol, United Kingdom
wl14928@bristol.ac.uk

Simon McIntosh-Smith
High-Performance Computing Group
University of Bristol
Bristol, United Kingdom
s.mcintosh-smith@bristol.ac.uk

Tom Deakin
High-Performance Computing Group
University of Bristol
Bristol, United Kingdom
tom.deakin@bristol.ac.uk

***Abstract*—With an ever-growing compute advantage over CPUs, GPUs are often used in workloads with ample BLAS computation to improve performance. However, several factors including data-to-compute ratio, amount of data re-use, and data structure shape can all impact performance. Hence, using a GPU is not a guarantee of better BLAS performance. In this work, we introduce the GPU BLAS Offload Benchmark (*GPU-BLOB*), a novel and portable benchmark that measures CPU and GPU compute performance of different BLAS kernels and problem configurations. From the *GPU offload threshold* (a BLAS kernel's minimum dimensions for a certain configuration where using a GPU is guaranteed to yield improved performance), we evaluate the per-node performance of three, in-production, HPC systems. We show that the *offload threshold* for GEMM is highly dependant on problem shape and number of consecutive BLAS calls, and that, contrary to conventional wisdom, GEMV *can* benefit from GPU acceleration, especially on SoC-based systems.**

***Index Terms*—BLAS, Performance, Heterogeneous, High-Performance Computing, Nvidia Grace-Hopper, AMD MI250X, Intel Ponte Vecchio**

I. Introduction & Motivation

Formally introduced as a Fortran library in 1979 by Lawson et al. [1], Basic Linear Algebra Subprograms (BLAS) are a set of low-level kernels used to compute common linear algebra operations. On top of the original library of vector-vector operations (Level 1), there have been two subsequent additions: matrix-vector operations (Level 2) [2]; and matrix-matrix operations (Level 3) [3]. Matrix-matrix multiplication (GEMM) and matrix-vector multiplication (GEMV) are arguably the most important of these BLAS kernels; used by a wide range of applications from scientific modelling[1] to Artificial Intelligence, and form the basis of many other BLAS kernels. As such, efficiently computing these operations can be paramount to good application performance.

Computing techniques such as Single Instruction Multiple Data (SIMD) lend themselves well to accelerating GEMM and GEMV computations given the inherent presence of (mathematical) vectors throughout. Being able to calculate multiple results concurrently improves computation throughput, and is why all modern processors have some form of vectorization capability. In x86 processors, instruction sets such as SSE, AVX(2), and AVX-512 bring extensive support for many different vector operations. Likewise, Arm processors have the NEON and SVE(2) vector instruction sets.

Over the last decade, it has become common to see the use of GPUs to improve application performance, largely due to their immense parallel compute capabilities. Looking at the Top500 over a seven year period, it is clear that we are in an increasingly heterogeneous world, where the number of high-performance computing systems with accelerators in the Top500 has risen from 91 in June 2017 to 193 in June 2024[2] [7]. Whilst many different types of applications benefit from offloading computation to the GPU, the resurgence of AI-type workloads and their reliance on GEMM computations [8] has led to the production of dedicated matrix engines on GPU devices. Examples include NVIDIA's Tensor cores [9], Intel's

CC-BY. This work was supported by the Engineering and Physical Sciences Research Council and Arm Ltd.

[1]Whilst Level 2 and 3 BLAS operations are commonly used in scientific applications, it should be noted that the time spent executing them is often not the dominating factor when executing at scale [4], [5].

[2]Although Deakin et al. noted that whilst this is a slowly growing fraction in terms of the number of GPU-accelerated machines in the list [6], the majority of the aggregated compute power on the list comes from the extremely large GPU-enabled machines.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00188

XMX matrix engines [10], and AMD's Matrix Cores [11], each promising substantial FLOP/s improvements over traditional GPU processing cores. This trend has continued onto the CPU, with new instruction set extensions such as Intel's AMX [10], IBM's MMA [12], Apple's AMX [13], and Arm's SME [14] providing ways to significantly improve GEMM and GEMV performance compared to *traditional* SIMD techniques without using an external GPU.

Common bottlenecks present across the majority of workload types are memory bandwidth/latency and CPU-GPU interconnect latency, caused by the increasing amount of data moved and used by modern applications. As such, another recent hardware trend is the production of System on Chip (SoC) designs; with CPUs, main memory, GPUs, etc. all present on the same chip, or tightly-integrated designs in a similar manner. Such SoCs generally yield much lower interconnect latencies due to the reduced physical distance between components. Recent examples of SoC designs include NVIDIA's Grace-Hopper Superchips, boasting a bespoke CPU-GPU interconnect (NVLink C2C) that is capable of 900 GB/s of bi-directional bandwidth [15]; and AMD's MI300A, with a single, unified address space allowing the CPU and GPU both to access all available memory at a peak bandwidth of 5.3 TB/s [16].

Thus, given the rapidly changing hardware landscape, we believe it critical to re-assess the GEMM and GEMV performance profiles for a range of systems with the latest hardware. Throughout this work, we test a variety of problem configurations to assess a) whether the common mantra of performing GEMM on GPU and GEMV on CPU still holds true, and b) if SoC style devices change the way we should approach GPU utilization for GEMM and GEMV kernels.

II. Related Work

There has been previous work to understand and optimize the behaviors of BLAS on CPUs and GPUs in heterogeneous systems, exploring how best to exploit both the CPU and the GPU in order to run BLAS kernels most efficiently. The MAGMA project [17] provides a collection of linear algebra libraries that combines the strength of the multi-core CPU and GPU architectures of heterogeneous systems to outperform libraries for the individual components of the systems taken separately [18], [19]. Dongarra et al. investigated the utility of the proposed batched BLAS extension to BLAS [20], comparing the compute performance of GPUs and CPUs for batched GEMM kernels, and show that using an interleaved memory layout results in batched BLAS running approximately twice as fast on a heterogeneous system.

There is also a body of work investigating which problems are better suited to a CPU and which are better suited to a GPU. For instance, Chikin et al. [21] build a selector to determine whether a program containing BLAS operations should be run on the CPU or the GPU. This selector is based on analytical models for both CPU and GPU performance, and indicates whether a significant performance gain is possible by analytically modelling the specific architecture of a given system. However, as an analytical model, this selector does not quantify differences between systems and would require a new model for each new architecture. Whilst an empirical benchmark, such as GPU-BLOB, allows for the quantification of such inter-system differences and can more easily measure the performance of new architectures due to its portability.

Beaumont et al. looked into offloading strategies specifically for computing forward activations in neural network training. Although determining exactly which calls should be offloaded to a GPU is an NP-complete problem, they were able to identify two heuristics by relaxing the problem [22].

Li et al. detail a performance comparison between CPUs and GPUs for BLAS of varying problem sizes [23]. This comparison showed that for each of DOT, GEMV, GEMM and TRSV, the GPUs achieved higher performance. For TRSM, the picture is more complex as for small vector sizes the CPUs were quicker than the GPUs (for larger vector sizes, the GPUs were again faster than the CPUs). However, these results are based on computation times and so do not include the critically important data transfer time for GPU measurements.

A runtime and energy efficiency comparison of a CPU and FPGAs for GEMV, GEMM, and SpMV was conducted by Favaro et al. [24]. Here, they showed that even when FPGAs had a longer runtime, they were more energy efficient.

Torres et al. investigated the runtime and energy consumption performance differences in the algorithms used for square SGEMMs in a range of BLAS libraries focusing on MKL, cuBLAS, and SYCL; across a number of heterogeneous systems using Intel CPUs and Nvidia GPUs [25]. They found that Intel Xeon Platinum 8489+ CPUs with MKL showed better time results than the NVIDIA A100 GPU with cuBLAS, though there was a greater loss of accuracy. However, unlike the benchmark presented here, Torres et al. did not investigate non-square matrices, and did not explore changes to this behaviour as problem size is varied.

Castelló et al. conducted a performance comparison between different GEMM algorithms based on the GEMM implementation in the BLIS library [26]. These algorithms place each of the operation's matrices into different levels of the memory hierarchy. Analyzing square and non-square GEMM problems, the authors found that some algorithms favored the square problem whilst others the non-square problems; highlighting the importance of using non-square GEMM problems when evaluating BLAS library performance. However, the non-square problems evaluated were limited to those with two dimensions fixed to 3000 or 100 in different combinations.

In view of related work in the literature; there remains a need for tools to determine, for a given heterogeneous system, at what shape and size a BLAS kernel is better handled by the GPU than the CPU. Hence, the work presented here.

III. Benchmark Overview

For this work we develop a new benchmark: the GPU BLAS Offload Benchmark, or *GPU-BLOB* [27]. The main design goals of GPU-BLOB are portability, such that it can be run easily on a wide range of systems; diversity, such

TABLE I
SGEMM RUN-TIMES (100 ITERATIONS) FOR DIFFERENT DEVICES AND BLAS LIBRARIES, VARYING THE VALUES OF $Alpha$ (α) AND $Beta$ (β). ALL RUN-TIMES ARE AN AVERAGE OF THREE RUNS. CPU RUNS ARE SINGLE THREADED.

BLAS Library	**Device**	**M**	**N**	**K**	$\alpha=1$ $\beta=0$	$\alpha=4$ $\beta=0$	$\alpha=1$ $\beta=2$
cuBLAS 24.3	NVIDIA® A100 40GB SXM	8,192	8,192	4	39.53 ms	39.23 ms	62.02 ms
rocBLAS 5.2.3	AMD® MI250X	8,192	8,192	4	188.64 ms	188.35 ms	210.46 ms
oneMKL 2024.1.0	Intel® Data Center GPU Max 1550	8,192	8,192	4	33.34 ms	32.99 ms	57.78 ms
oneMKL 2024.1.0	Intel Xeon Platinum 8468	8,192	8,192	4	2,307.38 ms	2,350.17 ms	3,137.10 ms
AOCL 4.2	AMD EPYC 7543P	8,192	8,192	4	6,833.02 ms	6,756.72 ms	9,175.32 ms

that it collects a large amount of performance data for a wide range of problem types; and completeness, such that it produces a *GPU offload threshold* for each BLAS kernel and problem type pairing. For each problem type (outlined in Section III-C), every possible combination of the given BLAS kernel's dimensions that adhere to the problem type's definition is executed iteratively. This allows GPU-BLOB to systematically measure the performance of all problem sizes between an upper and lower limit and accurately determine the *offload threshold*. These upper and lower limits are controlled through minimum and maximum dimension values provided by the user through the runtime arguments s and d respectively.

Whilst GPU-BLOB is typically built with both a CPU and GPU BLAS library, where each problem type and size is executed on the CPU then the GPU in an interleaved manner, it can also be built with either a CPU or a GPU library exclusively. This allows for individual component performance analysis, and is useful in situations where the two targeted BLAS libraries are incompatible or require different compilers.

A. BLAS Kernels

Whilst GPU-BLOB *can* support all BLAS kernels, this study focuses on GEMM and GEMV given that they form the basis of many other BLAS kernels and applications. For both, single and double-precision variants have been supported. All matrices and vectors are stored in column major format, with no transpositions performed. This gives GEMM leading dimensions of lda=M, ldb=K, and ldc=M, and GEMV vector increment values of $incx$=1 and $incy$=1.

Although for a study such as this one that assesses when offloading BLAS to a GPU is worthwhile, comparing the elapsed time would have been sufficient, GPU-BLOB calculates the GFLOP/s achieved by the CPU and GPU to allow it to also be used as a performance benchmark. As such, it is important to calculate the GFLOP/s accurately. For a GEMM of $C = Alpha \cdot A \cdot B + Beta \cdot C$, calculating the number of FLOPs can be broken into the following steps:

- $A{\cdot}B$ = MxNxK Fused Multiply-Adds = $2MNK$ FLOPs
- $Alpha \cdot AB$ = MxN Multiplications = MN FLOPs
- $Beta \cdot C$ = MxN Multiplications = MN FLOPs
- $AB + C$ = MxN Additions = MN FLOPs

Giving a total FLOPs count of $2MNK$ + $3MN$. A similar situation for a GEMV of $y = Alpha \cdot A \cdot x + Beta \cdot y$ yields a total FLOPs count of $2MN$ + $3M$. Although the number of FLOPs performed for GEMM and GEMV can be approximated to $2MNK$ and $2MN$ [20], [28] so long as dimensions K or N are sufficiently large respectively, we do not make this approximation due to some of the evaluated problems not meeting this requirement.

When $Alpha$=1 and $Beta$=0, there are certain optimizations that *could* be made to the GEMM FLOPs calculation. When $Beta$=0, $Beta \cdot C$ and $AB + C$ can be omitted. Likewise, when $Alpha$=1 there is no need to perform $Alpha \cdot AB$. Whilst there is *some* evidence to suggest these optimizations are made by some libraries [29], there is no concrete, recent evidence to support this. Hence, to see if these optimizations are commonly made in modern BLAS libraries we evaluated the SGEMM runtime for five different libraries. Table I shows the results of these SGEMM runs. First, with $Alpha$=1 we see that the value of $Beta$ can make a significant difference to the runtime of GEMM. When $Beta$=0, there is a 1.2x to 1.7x speedup compared to when $Beta$=2. This suggests that $Beta \cdot C$ and $AB + C$ are not performed. Second, when $Beta$=0, the value of $Alpha$ makes little difference to the total SGEMM runtime, suggesting no optimizations are made based on its value. With an average runtime difference across devices and BLAS libraries of 1.0% when $Alpha$=1 and $Alpha$=4, we can safely attribute this variance to noise.

Based on these results, we see that the $Beta$=0 optimization does seem to be implemented in modern BLAS libraries. As such, GPU-BLOB calculates FLOPs for GEMM and GEMV as $2MNK + MN + qMN$ and $2MN + M + qM$ respectively; where q=0 if $Beta$=0, or q=2 otherwise. When calculating GFLOP/s, the total execution time is measured as the time taken to perform i iterations of the given BLAS kernel and problem type to completion, where i is an argument passed in at runtime. GPU time measurements also include the time taken to move data to and from the GPU.

B. BLAS Library Support

In order to make GPU-BLOB as portable as possible, all major vendor and open-source BLAS libraries are supported. To improve portability further, there is native support for six C++ compilers, motivated by their widespread use or requirement to use them when compiling GPU-BLOB with one of the supported BLAS libraries. These compilers are: GNU g++, LLVM clang++, Intel icpx, Arm armclang++, NVIDIA nvc++, and AMD hipcc.

For both CPU and GPU library implementations, the input data structures are initialised using `rand` after calling `srand`

with a constant seed. Using a constant seed ensures that, during each individual execution of GPU-BLOB, the CPU and GPU data structures of the same dimensions are always initialized with identical contents. This allows a simple checksum to be calculated to validate that the CPU and GPU BLAS libraries are producing the same solutions for any problem type or size. To account for any floating-point rounding error, a 0.1% margin of error is permitted in the checksum. The output data structure is initialised to 0 throughout.

1) CPU BLAS Libraries:

GPU-BLOB currently supports the following CPU libraries:

- AMD Optimized Compute Libraries (AOCL)
- Arm Performance Libraries (ArmPL)
- NVIDIA Performance Libraries (NVPL)
- OpenBLAS
- Intel oneAPI Math Kernel Library (oneMKL)

Each library has been implemented with the common Cblas interface, with the exception of AOCL, which, given it is based on BLIS [30], uses the non-standard BLIS interface.

All data structures are created using `malloc`, excepting oneMKL which uses Intel's bespoke `mkl_malloc` function with an alignment value of 64 in order to achieve better performance [31]. To ensure the compiler does not optimize away any library calls where the output data structures are not used after-the-fact, an external function with empty body, `consume(void* a, void* b, void* c)`, is defined and built at compile time as a separate shared object.

2) GPU BLAS Libraries:

The GPU BLAS libraries that are currently supported are:

- Intel oneAPI Math Kernel Library (oneMKL)
- NVIDIA cuBLAS
- AMD rocBLAS

For each GPU library, there are three different implementations: Transfer-Once, Transfer-Always, and Unified Shared Memory (USM). Transfer-Once attempts to characterise situations where there is high data re-use, copying the input data structures (matrices A, B, and C for GEMM; matrix A and vectors x and y for GEMV) to the GPU before all i iterations of the BLAS kernel, then copying the output data structure (matrix C for GEMM; vector y for GEMV) back from the GPU to the host after all i iterations have been performed. Transfer-Always is the opposite: the appropriate data structures are transferred to and from the GPU before and after each and every iteration, mimicking an application with accelerated BLAS interleaved with some other host-based compute phase.

For the cuBLAS and rocBLAS implementations of these offload types, `cudaMallocHost` and `hipHostMalloc` are used to take advantage of pinned memory and optimize data transfers to the GPU [32]. For oneMKL, `sycl::malloc_host` is used for non-USM implementations as it has the same properties as `cudaHostAlloc`. Whilst this is technically a USM construct that establishes a device accessible pointer on the host, we manually copy the data to and from the device via `sycl::queue::memcpy` and force queue synchronisation before proceeding further to ensure intended behavior. All USM implementations operate in a similar manner to Transfer-Once, but instead use `cudaMallocManaged`, `hipMallocManaged`, or `sycl::malloc_shared` directives and remove the need for manual data movement.

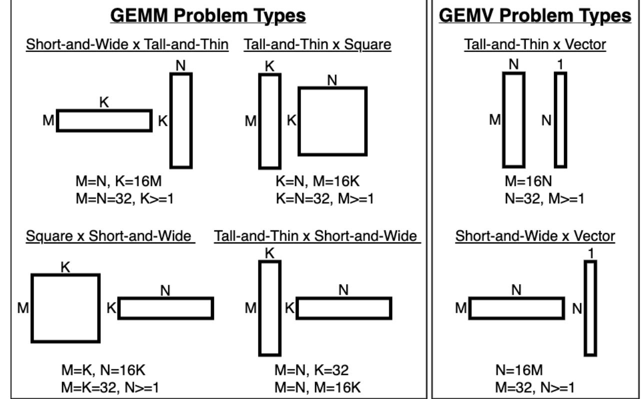


Fig. 1. GPU-BLOB non-square GEMM and GEMV problems.

C. Problem Types

In this work, we define a problem type as the fixed relationship between each of a BLAS kernel's specific dimensions. For example, a square problem type for GEMM would have the problem type definition of M=N=K. Whilst we commonly see only square GEMM and GEMV problems evaluated in the literature [18], [19], [23]–[25], when it comes to real applications and problems such as K-means clustering [33], LU factorization [34], and neural networks [35], [36], matrices of all shapes and sizes are used. Therefore, alongside the square problem type we have defined a varied set of non-square problems for GEMM and GEMV where (at least) one of the input matrices is rectangular. This provides a larger coverage of potential application domains and the rectangular shapes will help to highlight any irregular performance trends present within a BLAS library, especially in cases where one or more of the dimensions is very small or very large in comparison to the others. Fig. 1 provides an overview of the non-square GEMM and GEMV problem types evaluated by GPU-BLOB.

D. GPU Offload Threshold

The *GPU offload threshold* pinpoints the minimum dimension values for a given problem type and iteration count at which it becomes worthwhile to offload the BLAS computation to the GPU. This includes any overhead incurred by moving data to and from the GPU. For problem sizes greater than or equal to the identified *offload threshold*, the GPU is ensured to perform better than the CPU. By monitoring the performance of each problem size, GPU-BLOB is able to detect when the GPU begins to outperform the CPU for each data transfer type. To account for any momentary drops in GPU performance that are due to abnormal system behaviour

TABLE II
HARDWARE DETAILS FOR EACH NODE OF THE SYSTEMS USED IN THIS STUDY, ALONG WITH THE COMPILER AND BLAS LIBRARIES USED.

System	CPU	GPU	Compiler	CPU Library	GPU Library
DAWN	2x Intel Xeon Platinum 8468	4x Intel Data Center GPU Max 1550	icpx 2024.1.0	OneMKL 2024.1	OneMKL 2024.1
LUMI	1x AMD EPYC 7A53	4x AMD MI250X	g++ 7.5/hipcc 5.2.3	AOCL 4.1	rocBLAS 5.2.3
Isambard-AI	4x NVIDIA Grace-Hopper GH200 Superchip		nvc++ 24.5	NVPL 24.7	cuBLAS 24.5

or noise, the previous and current problem size's performance is taken into consideration. GPU-BLOB then monitors the performance for all subsequent problem sizes to ensure that the correct threshold has been identified.

The *offload threshold* provides a basis to help characterize which applications may benefit from offloading BLAS operations to the GPU on a specific system. By relating an application's matrix / vector shape and size to those evaluated by GPU-BLOB, configuring the iteration count to approximate the number of BLAS kernel computations, and relating the data movement characteristics to one of the data transfer types, a user can assess whether it would be worth porting their application to use a GPU for BLAS computations. Knowing this in advance can save the considerable amount of time and effort that porting an application requires in cases where a GPU provides no benefit. Throughout this study, we present an *offload threshold* as $\{m, n, k\}$ for GEMM and $\{m, n\}$ for GEMV; where m, n, and k are integers representing the value of dimensions M, N, and K respectively.

IV. EVALUATION

For this study, we ran GPU-BLOB across three systems: The University of Cambridge's DAWN [37], EuroHPC's LUMI [38], and The University of Bristol's Isambard-AI[3] [39]. Table II gives an overview of the node configuration, compiler(s) used, and the BLAS libraries targeted for each system. All builds of GPU-BLOB used optimization flags `-Ofast` and `-march=native` and each run used environment variables `OMP_PROC_BIND=close` (`spread` on LUMI due to multiple NUMA domains in a single CPU socket) and `OMP_PLACES=cores`. DAWN also used the environment variable `OMP_NUM_THREADS=48`, Isambard-AI `OMP_NUM_THREADS=72`, and LUMI `BLIS_NUM_THREADS=56`[4]; which ensured that on each system a full CPU socket was targeted[5]. Additionally, only a single GPU device was targeted on each system (on Isambard-AI this culminates in a single, whole, GH200 Superchip being used). Not using all available resources in each node is motivated by individual BLAS operations typically not being solved across sockets or devices, giving us a better representation of how BLAS is computed in real applications.

[3]Results for Isambard-AI were collected through an early access program and before full system acceptance.

[4]Despite differing recommendations from the AOCL user guide [30], we found these variables yielded the best performance on LUMI.

[5]Although each node of LUMI contains 64 physical cores, currently only 56 are available for use [40].

When targeting the GPU on DAWN and LUMI, special considerations were taken into account to ensure maximum performance. On DAWN, although each of the Intel 1550 GPU's contains two tiles [41], we target only one by enabling *Explicit Scaling*, as recommended, to avoid cross-tile communication costs [42]. On LUMI, each of the MI250X's is seen in software as two separate devices [43]. Again, GPU-BLOB only targets a single die and `#SBATCH --gpu-bind=closest` was used to pin the process to the NUMA node the target GPU is connected to, ensuring ideal host-device communication. To ensure USM can operate properly on LUMI, we use the `HSA_XNACK=1` environment variable to allow the GPU to signal page faults to the host. Not doing so forces all device accesses to host-resident memory to cross the host-device interconnect as no page migration occurs. Not using `HSA_XNACK=1` has been seen to cause a data-transfer performance penalty of up to 40x on an AMD MI100 [44].

Throughout this section, all runs, except those on LUMI, collect CPU and GPU data in an interleaved manner (GPU-BLOB's default execution style). This exception is due to the rocBLAS implementation of GPU-BLOB only being compatible with the hipcc compiler, as part of the library requires HIP constructs. When AOCL is used with this compiler, the CPU performance is extremely poor despite both being AMD products. Whilst surprising, AMD does not claim AOCL to be compatible with hipcc [30], and so GNU g++ 7.5 was used to compile LUMI's CPU version of GPU-BLOB instead.

On each system, GPU-BLOB uses a dimension size range of s=1 and d=$4,096$, ensuring a wide range of problem sizes are run for each BLAS kernel and problem type. Additionally, five iteration counts of varying sizes are used to mimic a concise range of data re-use patterns: 1, 8, 32, 64, 128.

A. Square GEMM

Table III shows that each system presents its own unique *offload threshold* characteristics for square GEMMs. Whilst DAWN sees very similar, moderate *offload threshold* values for all transfer types at one iteration, looking at Fig. 2 we can see a sharp CPU performance drop at $\{629, 629, 629\}$ that is gradually recovered from as the problem size increases. A similar performance drop also occurs for DGEMM. Without this drop, the one iteration square GEMM *offload thresholds* on DAWN would have likely been much higher; showcasing how large an impact BLAS library heuristics can have on performance. As the iteration count increases, this CPU performance drop has much less impact on the *offload threshold* due to much steeper Transfer-Once and USM performance curves. This leads to the marginally lower *offload thresholds*

TABLE III
SQUARE SGEMM:DGEMM (M=N=K) GPU OFFLOAD THRESHOLDS FOR EACH DATA TRANSFER TYPE AND HPC SYSTEM.

Iterations	DAWN			LUMI			Isambard-AI		
	Once	Always	USM	Once	Always	USM	Once	Always	USM
1	629 : 582	629 : 582	657 : 626	502 : 237	441 : 234	— : —	26 : 26	26 : 26	196 : 411
8	572 : 485	629 : 603	596 : 529	153 : 125	512 : 256	606 : 539	26 : 26	26 : 26	26 : 26
32	514 : 377	1018 : 833	509 : 389	2 : 2	512 : 461	442 : 256	26 : 26	26 : 26	26 : 26
64	514 : 361	1153 : 1153	465 : 436	2 : 2	589 : 961	381 : 239	26 : 26	26 : 26	26 : 26
128	514 : 361	1265 : 1153	412 : 377	2 : 2	512 : 1009	189 : 153	26 : 26	26 : 26	26 : 26

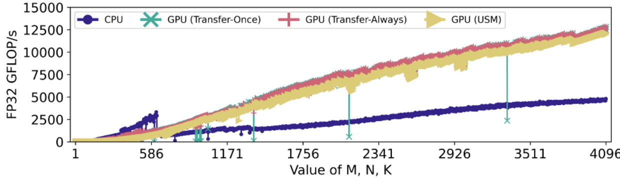
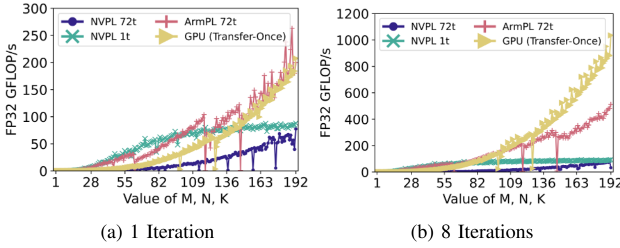


Fig. 2. Square SGEMM performance (1 iteration) on DAWN.



(a) 1 Iteration

(b) 8 Iterations

Fig. 3. Square SGEMM performance on Isambard-AI for different CPU libraries and configurations.

for SGEMM and substantially lower for DGEMM seen in Table III. For Transfer-Always, we see the *offload threshold* increase twofold by 128 iterations, which we expect as data is transferred to and from the GPU every iteration. Thus, the more iterations performed, the greater the GPU offload overhead. The same behaviour for Transfer-Always can also be seen on LUMI, although to a lesser extent for SGEMM.

Compared to DAWN, LUMI has two clear differences in its *offload threshold* profile. The first is the incredibly low Transfer-Once *offload threshold*; reaching {153, 153, 153} by eight iterations and stabilising at {2,2,2} from 32 iterations onwards. A contributing factor to this is the use of different CPU BLAS libraries, which, will inherently have different performance profiles and heuristics. Another is that each CPU socket in DAWN is considerably more powerful that those in LUMI: 1,536 FP64 FLOPs/cycle [45] vs. 896 FP64 FLOPs/cycle [43] respectively. This means DAWN has a smaller difference in CPU-GPU performance, and thus achieves larger *offload thresholds*. The other key difference between the systems is the USM *offload thresholds* relative to Transfer-Once. Whilst on DAWN, USM is on-par with Transfer-Once for all iteration counts, on LUMI, USM consistently has much higher *offload thresholds*. Given that data is transferred to and from the GPU at the same rate, this poor USM performance must be a result of the vendor's page migration heuristics.

Given its very capable CPU, with a theoretical peak of 1,152 FP64 FLOPs/cycle [46], Isambard-AI demonstrates that the SoC-based design of the GH200 Superchip almost entirely amortises the data transfer overhead of using a GPU for square GEMM; with an *offload threshold* of {26, 26, 26} across almost all iterations and transfer types. Similar to LUMI, USM performance lags behind the Transfer-Once performance at one iteration, but this gap quickly closes as the iteration count increases. However, if we compare these 72-thread NVPL results with ArmPL 24.04 or single-threaded NVPL results, as seen in Fig. 3 for the first 192 problem sizes, it is evident that library heuristics are one cause of the extremely low *offload thresholds* seen on Isambard-AI. When one iteration is performed, both ArmPL and single-threaded NVPL perform considerably better than NVPL 72-threads for these small problem sizes. This indicates that NVPL seemingly attempts to use all available threads for every problem size, whilst ArmPL scales the thread count with the problem size to optimize performance. As the iteration count increases to eight, the same behaviour is seen, however, the additional work done allows the GPU to showcase its immense raw performance advantage and decrease the *offload threshold*.

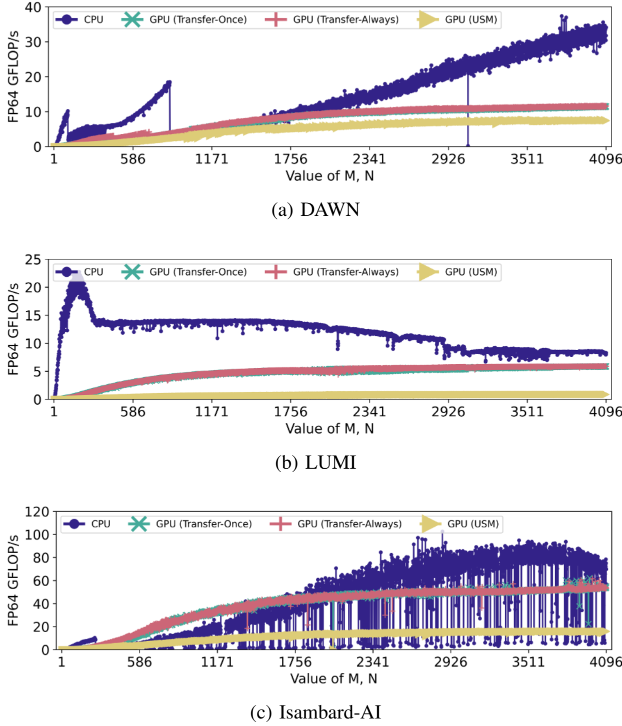
B. Square GEMV

Given a much higher data-to-compute ratio compared to square GEMMs, it is often thought that offloading GEMV computations to the GPU is not worthwhile. At one iteration, and all iterations for Transfer-Always (mimicking an application with low data re-use), we can see from Table IV that this is true for all systems as no *offload thresholds* are produced. However, the DGEMV performance curves seen in Fig. 4 show that on DAWN and Isambard-AI there is a considerable range of problem sizes where the GPU *does* outperform the CPU due to a CPU performance drop. This is also true for SGEMV on these systems at one iteration, where BLAS library heuristics are likely the cause of the stepped performance curves. Conversely, on LUMI we see from Fig. 4 that the CPU always outperforms the GPU at one iteration by a healthy (but narrowing) margin.

As the iteration count increases, Table IV shows that each system exhibits a unique *offload threshold* profile. Isambard-AI exhibits a very static *offload threshold* for Transfer-Once and USM, regardless of the iteration count. Analysis of Fig. 5

TABLE IV
SQUARE SGEMV:DGEMV (M=N) GPU OFFLOAD THRESHOLDS FOR EACH DATA TRANSFER TYPE AND HPC SYSTEM.

Iterations	DAWN			LUMI			Isambard-AI		
	Once	Always	USM	Once	Always	USM	Once	Always	USM
1	— : —	— : —	— : —	— : —	— : —	— : —	— : —	— : —	— : —
8	4089 : 3840	— : —	— : —	952 : 1197	— : —	— : —	256 : 256	— : —	— : —
32	4081 : 3065	— : —	4089 : 3521	569 : 617	— : —	2129 : 1885	256 : 249	— : —	256 : 255
64	3953 : 3065	— : —	4081 : 3361	529 : 601	— : —	1219 : 1205	256 : 249	— : —	256 : 251
128	4081 : 3321	— : —	4089 : 3481	465 : 545	— : —	754 : 909	256 : 249	— : —	256 : 249

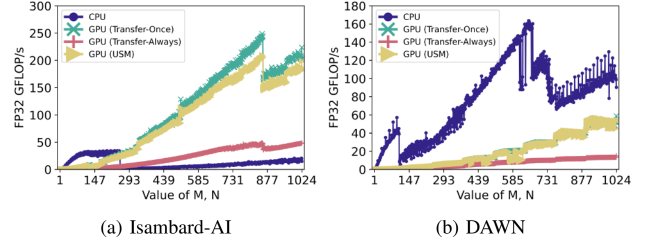


(a) DAWN

(b) LUMI

(c) Isambard-AI

Fig. 4. Square DGEMV performance (1 iteration).



(a) Isambard-AI

(b) DAWN

Fig. 5. Square SGEMV performance (128 iterations) on Isambard-AI and DAWN.

shows Isambard-AI to have very steep Transfer-Once and USM performance curves from fairly small problem sizes; a trend that is apparent from eight iterations onwards for SGEMV and DGEMV. Despite the visible CPU performance drop at approximately {256, 256} (which is consistent for all iteration counts), it is likely that the GPU would begin to outperform the CPU at close to this point anyway. In contrast, Fig. 5 shows DAWN to have much shallower and slowly increasing Transfer-Once and USM performance curves. Their small gradient, coupled with a better performing (and mature) CPU BLAS library, leads to the consistently high and static *offload thresholds* seen in Table IV for DAWN[6]. Thus, it is evident that the GH200's low CPU-GPU interconnect latency allows Isambard-AI's GPU to perform much better on memory-bound kernels, such as GEMV, compared to non-SoC based heterogeneous systems like DAWN.

Akin to the trends seen in Section IV-A, LUMI's GEMV *offload thresholds* steadily decrease as the iteration count increases; making it worthwhile to use the GPU at reducing problem sizes as data re-use increases. Given GEMV's high data-to-compute ratio, we would typically expect better CPU performance for all but the largest of problem sizes. Additionally, examining LUMI's GEMV performance across all benchmark runs we see that the CPU performance curve remains identical regardless of the number of iterations performed. Using `perf stat`, we discover that an SGEMV of M=N=2048 run for 1000 iterations uses only 0.89 CPUs. In comparison, an SGEMM of M=N=K=2048 run for 1000 iterations uses 50.2 CPUs. Hence, the poor GEMV performance achieved on LUMI is due to AOCL not parallelizing GEMV operations. Switching to OpenBLAS version 0.3.24 and using OMP_NUM_THREADS=56, we see significantly improved square GEMV performance across all iteration counts. Fig. 6 shows a comparison of AOCL and OpenBLAS performance for 128 iterations of square DGEMV on LUMI, where the performance difference is clear. Despite poorer small problem size performance, OpenBLAS produces no *offload thresholds* for any data transfer type across all iteration counts; solidifying how crucial BLAS library choice can be to performance and that vendor libraries are not always the best choice.

C. Non-Square GEMM

Across all eight non-square GEMM problem types evaluated, there are no consistent themes on all three systems, except that M=N, K=16M produces an SGEMM and DGEMM *offload threshold* on all systems at one iteration. On DAWN

[6]DGEMV sees lower *offload thresholds* than SGEMV due to a steady, shallow, CPU performance decrease that starts between M=N=3000 and M=N=3500 depending on iteration count.

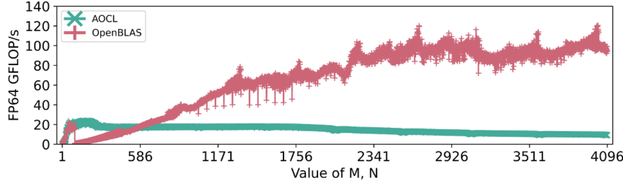


Fig. 6. AOCL (v4.1) vs. OpenBLAS (v0.3.24) square DGEMV CPU performance (128 iterations) on LUMI.

TABLE V
THE ITERATION COUNT AT WHICH EACH SGEMM:DGEMM NON-SQUARE PROBLEM TYPE FIRST YIELDS AN OFFLOAD THRESHOLD.

Problem Type		DAWN	LUMI	Isambard-AI
M=N	K=16M	1 : 1	1 : 1	1 : 1
M=N=32	$K \geq 1$	— : —	8 : —	1 : 1
K=N	M=16K	1 : 1	8 : 8	1 : 1
K=N=32	$M \geq 1$	— : —	32 : 8	1 : 1
M=K	N=16K	1 : 1	1 : 8	1 : 1
M=K=32	$N \geq 1$	— : —	32 : 32	1 : 1
M=N	K=32	8 : 8	32 : 32	8 : 8
M=N	M=16K	1 : 1	8 : 8	1 : 1

and LUMI at iteration counts greater than one, Transfer-Always was unable to produce an *offload threshold* for any problem type; whilst on Isambard-AI all transfer methods produced very similar *offload thresholds*. As such, for the remainder of this section we focus on Transfer-Once *offload thresholds*, given USM also transfers data once per problem size but can lag behind Transfer-Once's performance. Table V shows at which of the tested iteration counts did each non-square problem first produce an *offload threshold*.

On DAWN, we see that problem types where two of the dimensions are fixed to a (relatively) small value never yield an *offload threshold*. This is likely due to their much lower *Arithmetic Intensity* ($\frac{\text{FLOPs}}{\text{Bytes Used}}$) compared to the other non-square or square problem types, making data movement to the GPU a performance bottleneck. All other non-square problem types on DAWN yield (at varying sizes) an *offload threshold* at one iteration; excepting M=N, K=32 which also has a smaller *Arithmetic Intensity* due to its small fixed K dimension.

Analyzing non-square GEMM on LUMI, we see a large variation in the iteration count at which each non-square problem generates an *offload threshold*; with inconsistency also present between SGEMM and DGEMM runs of the same problem type. The most notable cases are M=N=32, $K \geq 1$ where DGEMM never produces an *offload threshold* but SGEMM does at an iteration count between one and eight; and K=N=32, $M \geq 1$ where SGEMM produces an *offload threshold* between eight and 32 iterations whilst DGEMM does so between one and eight. For the former problem type, we see SGEMM exhibit a large Transfer-Once GPU performance jump at {32, 32, 2560} for all iteration counts, whereas for DGEMM the GPU performance flat-lines at a low GFLOP/s value very early on. Hence, on LUMI, we cannot conclude (as we did for DAWN) that a lower *Arithmetic Intensity* equates to longer-lived or permanent CPU dominance due to irregular BLAS library heuristics. Rather, it seems that each individual problem type will non-deterministically perform better on the CPU or GPU.

Finally, on Isambard-AI it is again clear how the SoC design's reduced CPU-GPU latency facilitates improved GPU performance compared to the other systems. With all problem types yielding very low *offload* thresholds for all transfer types and iteration counts, there are few instances where offloading GEMM computations to the GPU is not worthwhile. Despite M=N, K=32 not producing an *offload threshold* until eight iterations, analysis of the one iteration performance graph shows behaviour similar to that seen in Fig. 4, whereby the quickly plateauing GPU performance and slowly increasing CPU performance leads to a large portion of small to mid-range problem sizes that perform better on the GPU.

TABLE VI
THE ITERATION COUNT AT WHICH EACH SGEMV:DGEMV NON-SQUARE PROBLEM TYPE FIRST YIELDS AN OFFLOAD THRESHOLD.

Problem Type		DAWN	LUMI	Isambard-AI
M=16N		— : —	8 : 8	1 : 1
N=32	$M \geq 1$	— : —	64 : 32	1 : 1
N=16M		— : —	— : —	1 : 1
M=32	$N \geq 1$	— : —	— : —	1 : 1

D. Non-Square GEMV

Like the results discussed in Section IV-C, Table VI shows non-square GEMV problems to have system-specific patterns of which iteration count the first *offload threshold* is present. On DAWN, it is clear from the results shown and from performance graph analysis that non-square GEMV problems are never worth offloading to the GPU. Despite this, it is worth noting that for N=16M and occasionally M=16N, the top 5%-10% of problem sizes saw approximately equivalent performance on CPU and GPU Transfer-Once / USM; SGEMV and DGEMV. Though, akin to that seen in Fig. 4, this is due to a sudden drop in CPU performance.

For LUMI, we see that non-square problems where the N dimension is considerably larger than M never yield an *offload threshold*. Analysis of the performance graphs confirms a commanding CPU performance lead over all GPU transfer types and iteration counts. Conversely, the M=16N problem sees a Transfer-Once *offload threshold* of {2464, 146} for SGEMV and {1760, 110} for DGEMV at eight iterations that decreases marginally as the iteration count increases. From 64 iterations, a USM *offload threshold* is produced, but is always far larger than Transfer-Once. Finally, for the N=32, $M \geq 1$ problem, we never see a Transfer-Always or USM *offload threshold*, with the Transfer-Once performance only overtaking the CPU at 32 and 64 iterations for DGEMV and SGEMV respectively. Starting at {3000, 32} and {3706, 32}, the DGEMV and SGEMV *offload thresholds* steadily decrease to {2155, 32} and {3200, 32} respectively by 128 iterations. Hence, on

LUMI using AOCL and rocBLAS, even with high amounts of data re-use only specific, moderate to large, non-square GEMV problem types benefit from GPU acceleration.

Despite Table VI aligning Isambard-AI's results with those seen before, where only the smallest of problem sizes are run best on the CPU, NVPL heuristics seem somewhat to blame for this. For problem types with one dimension fixed to the value of 32, the CPU performance is comfortably ahead of GPU performance until $\{2048, 32\}$ or $\{32, 2048\}$ where (again) we see a large performance drop off. Whilst this is present at all iteration counts, it is only at one iteration where it is directly to blame for the production of an *offload threshold*. For all other iterations, an early CPU performance plateau and steadily rising Transfer-Once performance makes it insignificant to the value of the *offload threshold*. Whilst all three transfer methods yield an *offload threshold* at all iteration counts for all non-square GEMV problems, for Transfer-Always this is aided by the CPU performance drop for these problem types. Additionally, USM is seen to under-perform Transfer-Once initially by 2x at one iteration, narrowing to almost identical performance by 64 iterations. But, again, this is only relevant for these specific problem types given that M=16N and N=16M have such small *offload thresholds* for all iteration counts and transfer types throughout.

V. Conclusions & Future Work

To conclude, using GPU-BLOB and the *GPU offload threshold* we have shown across three heterogeneous HPC systems that there are very few GEMM and GEMV problems that consistently benefit from GPU acceleration at a particular size. Whilst Isambard-AI's low CPU-GPU interconnect latency was able to (almost) completely amortise the GPU offload penalty in most cases, for non-SoC based systems, the results were not so simple.

Unsurprisingly, moderate or larger square GEMM problems do benefit from GPU acceleration on DAWN, but, we have seen on LUMI that when data re-use is high it can be beneficial to always use the GPU. When data re-use is low, the square GEMM *offload threshold* increases drastically with the iteration count on both systems. For square GEMV, we saw on DAWN that only very large problem sizes would benefit from a GPU. Whereas on LUMI, the square GEMV *offload threshold* was surprisingly low due to a lack of multi-threading in AOCL. Looking at non-square problems on DAWN, many GEMMs required a reasonably large problem size or more than one iteration to yield a Transfer-Once *offload threshold* (if at all), and all non-square GEMV problems failed to ever perform better on the GPU. On LUMI, most non-square GEMMs yielded an *offload threshold*, but at a higher iteration count. Whether a non-square GEMV yielded an *offload threshold* depended on the ratio between M and N and a high iteration count. The one consistency across all systems (including Isambard-AI) was square GEMV problems with low data re-use (i.e., Transfer-Always) never producing an *offload threshold*, regardless of the iteration count.

Hence, for non-SoC systems, whether the common mantra of "*performing GEMM on GPU and GEMV on CPU*" holds true depends entirely on multiple factors: from the hardware used, where a system may have a lower powered CPU to feed high performing GPUs; to BLAS library choice, where one library's heuristics may cause sudden performance drops, or another having sub-optimal multi-threading capabilities; to the shape of the specific problem, where a lower *Arithmetic Intensity* often leads to poorer GPU performance, but not always; to the amount of data re-use, where often low re-use causes a significant increase in the *GPU offload threshold*; to the data transfer methodology used, where we often saw the USM performance lag behind the Transfer-Once performance on Isambard-AI and LUMI. However, with new SoC style devices such as the NVIDIA GH200 found inside Isambard-AI, our GEMV-based mantra must change; with it being very rare to encounter a GEMM or GEMV problem that would not benefit from GPU acceleration.

Whilst the *GPU offload threshold* introduced in this study provides a useful means of determining whether a given BLAS operation could benefit from GPU acceleration, it was shown to have some limitations. First, as we saw in Fig. 4 the absence of an *offload threshold* does not equate to the CPU outperforming the GPU for *all* problem sizes. Second, the *offload threshold* alone does not indicate by how much the GPU outperforms the CPU. Hence, generating performance graphs is likely required to accurately determine whether a BLAS-based application would benefit from GPU acceleration, and whether enough performance can be gained for the code porting effort to be worthwhile.

Building on this work, we aim to analyse the impact of CPU matrix engines on the *offload threshold*. Given the different implementations from a variety of vendors, assessing their impact on GEMM and GEMV kernels on heterogeneous nodes is of particular interest. Given their prevalence in AI and mixed-precision computations, we are also looking to support half-precision kernels; FP16 and Bfloat16. Not all BLAS libraries support HGEMM, and some that do are not intuitive to use, so were not included in this initial study. oneMKL's C interface, for example, *does* have HGEMM support, but its data-type `MKL_F16` is defined internally as an `unsigned short` [47]. With no conversion functions (as of v2024.1.0) to or from `MKL_F16`, akin to those found in the CUDA API, it makes targeting half-precision kernels complex.

Investigation of the performance characteristics of batched kernels would expand the range of applications that GPU-BLOB can evaluate. Given that batched kernels can greatly improve GEMM performance for small problem sizes *if* many can be computed concurrently [48], [49], we wish to quantify the effect that this has on the *offload threshold*.

Finally, we are currently working to support sparse BLAS computations in GPU-BLOB. Again, this would broaden the scope of applications we can evaluate, but, due to the large variety of sparse problem types, narrowing this down into a core subset that is representative of a wide array of applications is non-trivial.

Acknowledgments

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/T022159/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk).

This work used the Isambard-AI UK National AI Service operated by The University of Bristol, and funded by STFC/UKRI and DSIT.

We acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI, hosted by CSC (Finland) and the LUMI consortium through a EuroHPC Regular Access call.

The University of Bristol is an Intel oneAPI Center of Excellence, which helped support this work.

References

[1] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Softw.*, vol. 5, p. 308–323, sep 1979.

[2] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, p. 1–17, mar 1988.

[3] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, p. 1–17, mar 1990.

[4] H. Waugh and S. McIntosh-Smith, "On the Use of BLAS Libraries in Modern Scientific Codes at Scale," in *Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI*, pp. 67–79, Springer International Publishing, 2020.

[5] J. Domke, E. Vatai, A. Drozd, P. ChenT, Y. Oyama, L. Zhang, S. Salaria, D. Mukunoki, A. Podobas, M. WahibT, and S. Matsuoka, "Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws?," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1056–1065, 2021.

[6] T. Deakin, J. Cownie, W.-C. Lin, and S. McIntosh-Smith, "Heterogeneous Programming for the Homogeneous Majority," in *International Workshop on Performance, Portability and Productivity in HPC held in conjunction with Supercomputing (P3HPC)*, IEEE, 2022.

[7] Top500, "LIST STATISTICS." https://www.top500.org/statistics/list/, 2024. Accessed: 18th July 2024.

[8] P. Warden, "Why GEMM is at the heart of deep learning." https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/, April 2015. Accessed: 18th July 2024.

[9] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA Tensor Core Programmability, Performance & Precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 522–531, 2018.

[10] A. Wolf, "Architecting for Accelerators – Intel AMX and Intel XMX." https://community.intel.com/t5/Blogs/Tech-Innovation/Artificial-Intelligence-AI/Architecting-for-Accelerators-Intel-AMX-and-Intel-XMX/post/1481022, April 2023. Accessed: 18th July 2024.

[11] G. Schieffer, D. Medeiros, J. Faj, A. Marathe, and I. Peng, "Characterizing the Performance, Power Efficiency, and Programmability of AMD Matrix Cores," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software*, 2024.

[12] W. J. Starke, B. W. Thompto, J. A. Stuecheli, and J. E. Moreira, "IBM's POWER10 Processor," *IEEE Micro*, vol. 41, no. 2, pp. 7–14, 2021.

[13] D. Johnson, "AMX: Apple Matrix coprocessor." https://gist.github.com/dougallj/7a75a3be1ec69ca550e7c36dc75e0d6f, September 2022. Accessed: 24th July 2024.

[14] F. Wilkinson and S. McIntosh-Smith, "An Initial Evaluation of Arm's Scalable Matrix Extension," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 135–140, 2022.

[15] NVIDIA, *NVIDIA GH200 Grace Hopper Superchip Architecture*. NVIDIA, 2024.

[16] A. Smith, G. H. Loh, M. J. Schulte, M. Ignatowski, S. Naffziger, M. Mantor, M. Fowler, N. Kalyanasundharam, V. Alla, N. Malaya, J. L. Greathouse, E. Chapman, and R. Swaminathan, "Realizing the AMD Exascale Heterogeneous Processor Vision," in *ISCA '24: Proceedings of the 51st Annual International Symposium on Computer Architecture*, Association for Computing Machinery, 2024.

[17] Innovative Computing Laboratory, "MAGMA." https://icl.utk.edu/magma/, 2024. Accessed: 23rd July 2024.

[18] R. Nath, S. Tomov, and J. Dongarra, "An Improved Magma Gemm For Fermi Graphics Processing Units," *The International Journal of High Performance Computing Applications*, vol. 24, pp. 511–515, November 2010.

[19] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, 2010.

[20] A. Abdelfattah, S. Tomov, and J. Dongarra, "Matrix multiplication on batches of small matrices in half and half-complex precisions," *Journal of Parallel and Distributed Computing*, vol. 145, pp. 188–201, 2020.

[21] A. Chikin, J. N. Amaral, K. Ali, and E. Tiotto, "Toward an Analytical Performance Model to Select between GPU and CPU Execution," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 353–362, 2019.

[22] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, "Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training," in *Euro-Par 2020: Parallel Processing* (M. Malawski and K. Rzadca, eds.), (Cham), pp. 151–166, Springer International Publishing, 2020.

[23] F. Li, Y. Ye, X. Zhang, and Z. Tian, "CPU versus GPU: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms," *Neural Computing and Applications*, vol. 31, pp. 4353–5365, 2019.

[24] F. Favaro, E. Dufrechou, J. Oliver, and P. Ezzatti, "Evaluation of dense and sparse linear algebra kernels in FPGAs," 04 2024.

[25] L. A. Torres, C. J. B. H, and Y. Denneulin, "Evaluation of computational and energy performance in matrix multiplication algorithms on CPU and GPU using MKL, cuBLAS and SYCL," 2024.

[26] A. Castelló, E. S. Quintana-Ortí, and F. D. Igual, "Anatomy of the blis family of algorithms for matrix multiplication," in *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 92–99, 2022.

[27] F. Wilkinson and A. Cockrean, "UoB-HPC/GPU-BLAS-Offload-Benchmark: GPU-BLOB v1.0.0," Sept. 2024.

[28] NVIDIA, "Matrix Multiplication Background User's Guide." https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, Feb 2023. Accessed: 18th July 2024.

[29] Bishwa, "FLOPS calculation in cublasDgemm." https://forums.developer.nvidia.com/t/flops-calculation-in-cublasdgemm/24673/5, October 2011. Accessed: 16th July 2024.

[30] Advanced Micro Devices, *AOCL User Guide: Revision 4.2*. Advanced Micro Devices, feb 2024.

[31] Intel Corporation, "Data Alignment and Leading Dimensions." https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide-linux/2024-1/coding-techniques.html, 2024. Accessed: 19th July 2024.

[32] M. Harris, "How to Optimize Data Transfers in CUDA C/C++." https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/, December 2012. Accessed: 19th July 2024.

[33] I. S. Dhillon, Y. Guan, and B. Kulis, "Kernel k-means: spectral clustering and normalized cuts," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, (New York, NY, USA), p. 551–556, Association for Computing Machinery, 2004.

[34] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched gemm for gpus," in *High Performance Computing* (J. M. Kunkel, P. Balaji, and J. Dongarra, eds.), (Cham), pp. 21–38, Springer International Publishing, 2016.

[35] S. Yin, Q. Wang, R. Hao, T. Zhou, S. Mei, and J. Liu, “Optimizing irregular-shaped matrix-matrix multiplication on multi-core dsps,” in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, (Los Alamitos, CA, USA), pp. 451–461, IEEE Computer Society, sep 2022.
[36] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, “Libshalom: optimizing small and irregular-shaped matrix multiplications on armv8 multi-cores,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’21, (New York, NY, USA), Association for Computing Machinery, 2021.
[37] L. Walsh, “The rise of Dawn.” https://www.cam.ac.uk/stories/ai-supercomputer-dawn-research-energy-medicine-climate, February 2024. Accessed: 25th July 2024.
[38] P. Manninen, F. Robertsén, and G. Markomanolis, “May we introduce: LUMI.” https://www.lumi-supercomputer.eu/may-we-introduce-lumi/, November 2020. Accessed: 25th July 2024.
[39] S. Mcintosh-Smith, S. Alam, and C. Woods, “Isambard-AI: a leadership class supercomputer optimised specifically for Artificial Intelligence,” in *CUG 2024: Cray User Group*, May 2024.
[40] EuroHPC, “LUMI-G example batch scripts.” https://docs.lumi-supercomputer.eu/runjobs/scheduled-jobs/lumig-job/, 2024. Accessed: 15th July 2024.
[41] D. Mulnix, “Intel Data Center GPU Max Series Technical Overview.” https://www.intel.com/content/www/us/en/developer/articles/technical/intel-data-center-gpu-max-series-overview.html, April 2024. Accessed: 16th July 2024.
[42] J. R. Reinders, “Options for using a GPU Tile Hierarchy.” https://www.intel.com/content/www/us/en/developer/articles/technical/flattening-gpu-tile-hierarchy.html, October 2023. Accessed: 16th July 2024.
[43] EuroHPC, “GPU nodes - LUMI-G.” https://docs.lumi-supercomputer.eu/hardware/lumig/, 2024. Accessed: 15th July 2024.
[44] W.-C. Lin, S. McIntosh-Smith, and T. Deakin, “Preliminary report: Initial evaluation of StdPar implementations on AMD GPUs for HPC,” 2024.
[45] Intel Corporation, “Intel Xeon Platinum 8468 Processor.” https://www.intel.com/content/www/us/en/products/sku/231735/intel-xeon-platinum-8468-processor-105m-cache-2-10-ghz/specifications.html, 2023. Accessed: 17th July 2024.
[46] J. Evans, I. Finder, I. Goldwasser, J. Linford, V. Mehta, D. Ruiz, and M. Wagner, “NVIDIA Grace CPU Superchip Architecture In Depth.” https://developer.nvidia.com/blog/nvidia-grace-cpu-superchip-architecture-in-depth/, Jan 2024. Accessed: 23rd July 2024.
[47] Intel oneAPI Math Kernel Library, “mkl_types.h,” 2024. Line 107, Version 2024.1.0.
[48] C. Cecka, “Pro Tip: cuBLAS Strided Batched Matrix Multiply,” *NVIDIA Technical Blog*, Feb 2017. Accessed: 8th July 2024.
[49] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, “The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems,” *Procedia Computer Science*, vol. 108, pp. 495–504, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

Appendix A

On DAWN, explicit scaling, single-tile operation, and driver selection for the Intel Data Centre GPU Max 1550 was achieved using the following environment variables: `EnableImplicitScaling=0`; `ZE_FLAT_DEVICE_HIERARCHY=FLAT`; `ZE_AFFINITY_MASK=0,1,2,3,4,5,6,7`; `ONEAPI_DEVICE_SE- LECTOR="level_zero:0"`. Fig. 7 shows a comparison of oneMKL SGEMM Transfer-Once performance on DAWN using *Implicit Scaling* (viewing the GPU as one device) and *Explicit Scaling* (viewing the GPU as two separate devices). Clearly, implicit scaling yields much lower and less-consistent performance than explicit scaling, despite having twice the compute resources.

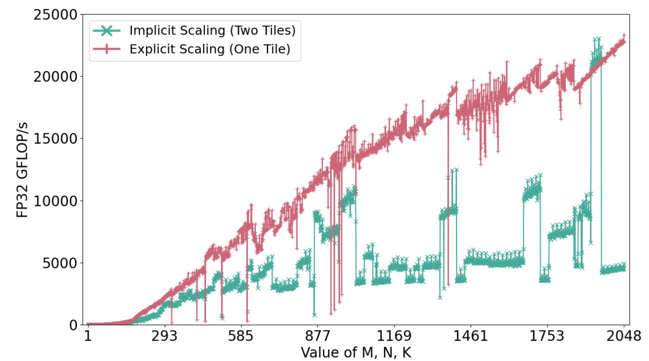


Fig. 7. DAWN GPU SGEMM performance (32 iterations) using implicit and explicit hardware scaling.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 A new portable benchmark, the GPU BLAS Offload Benchmark (GPU-BLOB), to facilitate the collection of CPU and GPU BLAS performance data on heterogeneous systems.
- C_2 A new metric, the *GPU Offload Threshold*, which, for a given BLAS problem type (or relationship between the dimensions) gives the dimension values at which the GPU is guaranteed to perform better than the CPU for all larger problem sizes.
- C_3 Data on when GEMM and GEMV problems of a certain size and shape will perform better on a GPU compared to a single CPU socket for three HPC systems: DAWN, LUMI, and Isambard-AI.
- C_4 An analysis on how and how often data is moved to and from the GPU affects the achieved GEMM and GEMV performance for three main GPU vendors and their respective BLAS libraries. This includes data for Unified Shared Memory (USM).

B. Computational Artifacts

A_1 GPU BLAS Offload Benchmark
10.5281/zenodo.13835297

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3, C_4	Tables 3-6 Figures 2-7

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

The artifact A_1 is contribution C_1.
One of the main outputs of artifact A_1 is the *GPU offload threshold* which is contribution C_2. This metric, in combination with A_1's output performance data, is used to formulate contribution C_3.
Artifact A_1 is developed to perform three data offload paradigms (Transfer-Once, Transfer-Always, USM). The performance data and *offload threshold* collected by A_1 is used to formulate contribution C_4.

Expected Results

The output of running GPU-BLOB on each of the three systems should yield the following results for contribution C_3:

- **Square GEMM**: Isambard-AI should have the lowest *offload threshold* at all iteration counts, followed by LUMI, then DAWN. Transfer-Once and USM *offload thresholds* should decrease as iteration counts increase on LUMI and DAWN; Transfer-Always should see the *offload threshold* increase as the iteration count increases.
- **Square GEMV**: On all three systems Transfer-Always should not yield an *offload threshold* at any iteration count. Transfer-Once and USM should also not yield an *offload threshold* at one iteration on any system. On DAWN and Isambard-AI, Transfer-Once and USM *offload thresholds* should remain somewhat constant regardless of iteration count. On LUMI, the *offload threshold* should decrease as iteration count increases.
- **Non-Square GEMM**: Isambard-AI should yield an *offload threshold* for each problem type at one iteration, excepting problem $\{M{=}N,\ K{=}32\}$. On DAWN, all problem types that fix two GEMM dimensions to 32 should never yield an *offload threshold*. All other problems (again excepting $\{M{=}N,\ K{=}32\}$) should yield an *offload threshold* at one iteration. Results on LUMI should not see a specific pattern, with most problem types yielding an *offload threshold* at eight or 32 iterations.
- **Non-Square GEMV**: No *offload threshold* should be seen for any problem type on DAWN, or problem types $\{N{=}16M\}$ or $\{M{=}32,\ N \geq 1\}$ on LUMI. Isambard-AI should yield an *offload threshold* for all problem types at one iteration.

For contribution C_4, we see the following behaviour on each of the three systems:

- **DAWN**: Transfer-Once and USM results should be approximately the same throughout. Transfer-Always always lags behind the other transfer types in terms of achieved performance except at one iteration where they should be equivalent.
- **LUMI**: Transfer-Always should see very similar results to Transfer-Once at one iteration, and then severely lag behind performance wise for all other iteration counts. USM results should be between Transfer-Once and Transfer-Always at all iteration counts.
- **Isambard-AI**: Transfer-Always should see very similar results to Transfer-Once at one iteration, and then severely lag behind performance wise for all other iteration counts. USM should lag behind Transfer-Once at lower (one and eight) iteration counts, but then be approximately equivalent for all other iteration counts.

Expected Reproduction Time (in Minutes)

Compilation of the benchmark should take up to 5 minutes depending on the system, given the required modules are available on the system. If they are not, then this can be expected to take 30 minutes per system.

Assuming all five experiments (i.e one experiment for each iteration value $i \in \{1, 8, 32, 64, 128\}$) of the benchmark can be run in parallel on each HPC system, artifact execution should take up to 8 hours (480 minutes) on DAWN and Isambard-AI, and 14 hours (840 minutes) on LUMI.

Artifact analysis (including generating graphs from the benchmark's performance data CSV files using the included Python script) for all three systems is expected to take 6 hours (360 minutes).

Artifact Setup (incl. Inputs)

Hardware: The following hardware is required for each of the systems evaluated in this study (denoted S_n):

- S_1 **DAWN**: Intel Xeon Platinum 8468 + Intel Data Center GPU Max 1550
- S_2 **LUMI**: AMD EPYC 7A53 + AMD MI250X
- S_3 **Isambard-AI**: NVIDIA Grace-Hopper GH200 Superchip (120GB LPDDR5X + 96GB HBM3)

Software: All systems require the GPU BLAS Offload benchmark V1.0.0; artifact A_1: https://github.com/UoB-HPC/GPU-BLAS-Offload-Benchmark/tree/v1.0.0

Each corresponding system S_n requires the following software:

S_1
- Intel oneMKL version 2024.1.0 via Intel oneAPI Bast Toolkit https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-download.html.

  NB Intel has removed support for versions 2024.2.0 and earlier as of 23rd September 2024.

S_2
- AOCL version 4.1 https://www.amd.com/en/developer/aocl/eula/aocl-4-1-eula.html?filename=aocl-linux-gcc-4.1.0.tar.gz
- rocBLAS via rocm version 5.2.3 https://rocm.docs.amd.com/en/docs-5.2.3/index.html

S_3
- NVPL version 24.7 https://developer.nvidia.com/nvpl-downloads
- cuBLAS version 24.5 via NVHPC version 24.5 https://developer.nvidia.com/nvidia-hpc-sdk-245-downloads

Datasets / Inputs: All data is generated by the GPU BLAS Offload Benchmark at runtime. See *Installation and Deployment* on how to generate the data sets used in this study.

Installation and Deployment: The benchmark is compiled, deployed, and executed on each of the corresponding systems S_n as follows. This collects all data used in this study:

S_1
- Compiler: icpx version 2024.1.0
- Compilation Command: make COMPILER=INTEL CPU_LIB=ONEMKL GPU_LIB=ONEMKL
- Execution: OMP_NUM_THREADS=48 OMP_PROC_BIND=close OMP_PLACES=cores ./gpu-blob -i n -s 1 -d 4096 where $n \in \{1, 8, 32, 64, 128\}$

S_2
- Two versions of the benchmark need to be compiled and executed due to compiler-library incompatibility: a CPU-only and a GPU-only version.
- CPU-only Compiler: g++ version 7.5
- CPU-only Compilation Command: make COMPILER=GNU CPU_LIB=AOCL CXXFLAGS="-L<AOCL_DIR>/lib -I<AOCL_DIR>/include/blis -Wl,-rpath,<AOCL_DIR>/lib"
- CPU-only Execution: BLIS_NUM_THREADS=56 OMP_PROC_BIND=close OMP_PLACES=cores ./gpu-blob_cpu -i n -s 1 -d 4096 where $n \in \{1, 8, 32, 64, 128\}$
- GPU-only Compiler: hipcc version 5.2.3
- GPU-only Compilation Command: make COMPILER=HIP CPU_LIB=ROCBLAS
- GPU-only Execution: ./gpu-blob_gpu -i n -s 1 -d 4096 where $n \in \{1, 8, 32, 64, 128\}$

S_3
- Compiler: nvc++ version 24.5
- Compilation Command: make COMPILER=NVIDIA CPU_LIB=NVPL GPU_LIB=CUBLAS CXXFLAGS="-L<NVPL_DIR>/lib -I<NVPL_DIR>/include -Wl,-rpath,<NVPL_DIR>/lib"
- Execution: OMP_NUM_THREADS=72 OMP_PROC_BIND=close OMP_PLACES=cores ./gpu-blob -i n -s 1 -d 4096 where $n \in \{1, 8, 32, 64, 128\}$

Artifact Execution

- T_1 Generate dataset using the execution command(s) seen in *Artifact Setup - Installation and Deployment*.
- T_2 Extract *offload thresholds* via:
  - S_1 or S_3: Taking *offload thresholds* from the tables printed to stdout by the benchmark.
  - S_2: Use the included calculateOffloadThreshold.py Python script on each pair of output CSV files (CPU and GPU CSV for the same problem type).
- T_3 Generate performance graphs using the included createGflopsGraphs.py Python script with the input being a CSV directory/
  - When processing results from S_2, each problem type's CPU-only and GPU-only CSV result files will need to be concatenated (removing the additional header row) prior to using the graphing Python script.

Task dependencies: $T_1 \rightarrow T_2$ and $T_1 \rightarrow T_3$.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

Per-system library compilation:

- DAWN:
  - Download Intel oneMKL version 2024.1.0 via Intel oneAPI Bast Toolkit from https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-download.html, selecting *Linux* and *Offline Installer*, then following the download instructions.
  - Run the install script via sh ./l_BaseKit_p_2024.1.0.100_offline.sh -a --cli --eula accept
  - Once installed, run source <oneapi_install_dir>/2024.1.0/setvars.sh to add the libraries and compilers to the relevant paths and be available for use.
- LUMI:
  - Download AOCL version 4.1 from https://www.amd.com/en/developer/aocl/eula/aocl-4-1-eula.html?filename=aocl-linux-gcc-4.1.0.tar.gz.
  - Download rocm version 5.2.3 from https://rocm.docs.amd.com/en/docs-5.2.3/index.html.
  - Install rocm using the instructions at https://rocm.docs.amd.com/projects/install-on-linux/en/latest/install/quick-start.html.
  - Run module load rocm/5.2.3 to make the rocm and rocBLAS libraries, and the hipcc compiler available for use.
- Isambard-AI:
  - Download NVPL version 24.7 from https://developer.nvidia.com/nvpl-downloads
  - Download NVHPC version 24.5 from https://developer.nvidia.com/nvidia-hpc-sdk-245-downloads
  - Install NVHPC by running nvhpc_2024_245_Linux_aarch64_cuda_12.4/install and following the script instructions.
  - Run module use NVHPC_24.5/modulefiles then module load nvhpc/24.5 to make nvc++ and cuBLAS available for use.

Per-system benchmark compilation:

- DAWN:
  - make COMPILER=INTEL CPU_LIB=ONEMKL GPU_LIB=ONEMKL
- LUMI:
  - CPU-only: make COMPILER=GNU CPU_LIB=AOCL CXXFLAGS="-L<AOCL_DIR>/lib -I<AOCL_DIR>/include/blis -Wl,-rpath,<AOCL_DIR>/lib"
  - GPU-only: make COMPILER=HIP CPU_LIB=ROCBLAS
- Isambard-AI:
  - make COMPILER=NVIDIA CPU_LIB=NVPL GPU_LIB=CUBLAS CXXFLAGS="-L<NVPL_DIR>/lib -I<NVPL_DIR>/include -Wl,-rpath,<NVPL_DIR>/lib"

Per-system code deployment guidelines:

- DAWN:
  - pvc partition, 1 node, 48 tasks per node, exclusive node access
  - Ensure to set the following environment variables to enable explicit GPU scaling and target only one of the two GPU tiles: NEOReadDebugKeys=1 PrintDebugSettings=1 ReturnSubDevicesAsApiDevices=1 EnableImplicitScaling=0 ZE_AFFINITY_MASK=0,1,2,3,4,5,6,7 ZE_FLAT_DEVICE_HIERARCHY=FLAT ONEAPI_DEVICE_SELECTOR="level_zero:0"
- LUMI:
  - CPU-only: standard-g partition, 1 node, 56 task per node, exclusive node access
  - GPU-only: standard-g partition, 1 node, 1 gpu per node, gpu-bind closest, 1 task per node, exclusive node access
  - For GPU-only, ensure to set the following environment variable: export HSA_XNACK=1
- Isambard-AI:
  - 1 node, 72 tasks per node, exclusive node access

Artifact Execution

To generate performance and *offload threshold* data for a given system, the following commands should be run:

- DAWN: OMP_NUM_THREADS=48 OMP_PROC_BIND=close OMP_PLACES=cores ./gpu-blob -i n -s 1 -d 4096 where $n \in \{1, 8, 32, 64, 128\}$
- LUMI: Two versions of the benchmark need to be compiled and executed due to compiler-library incompatibility:
  - CPU-only Execution: BLIS_NUM_THREADS=56 OMP_PROC_BIND=close OMP_PLACES=cores ./gpu-blob_cpu -i n -s 1 -d 4096 where $n \in \{1, 8, 32, 64, 128\}$
  - GPU-only Execution: ./gpu-blob_gpu -i n -s 1 -d 4096 where $n \in \{1, 8, 32, 64, 128\}$
- Isambard-AI: OMP_NUM_THREADS=72 OMP_PROC_BIND=close OMP_PLACES=cores

```
./gpu-blob -i n -s 1 -d 4096
```
where $n \in \{1, 8, 32, 64, 128\}$

Following this execution, *offload threshold* data for DAWN and Isambard-AI will be easily available given it is printed to `stdout` by the benchmark. For LUMI, the `calculateOffloadThreshold.py` Python script included with GPU-BLOB can be used on each pair of output CSV files (CPU and GPU CSV for the same problem type) to yield the *offload thresholds*.

Lastly, performance graphs of each system can also be generated using the data produced by GPU-BLOB. This can be done using the `createGflopsGraphs.py` Python script included with GPU-BLOB, where the single input argument is a single problem's CSV file directory. For LUMI, a pre-processing step of concatenating each problem type's CPU-only and GPU-only CSV result files (removing the additional header row) is required prior to using the graphing Python script.

Artifact Analysis (incl. Outputs)

From each system, one should expect the benchmark to have produced 28 CSV files per experiment run: 9 SGEMM, 9 DGEMM, 5 SGEMV, 5 DGEMV; one CSV file per problem type. These CSV files contain the raw GFLOP/s performance data for each problem size evaluated, along with problem dimensions, runtime, and other metrics. For DAWN and Isambard-AI (or when GPU-BLOB is run with a CPU *and* a GPU library), the *offload threshold* for each data transfer type and problem type will be printed in a table to `stdout`. For LUMI (or when CPU and GPU data is collected individually), the method for generating the *offload threshold* per problem type described above in *Artifact Execution* should be followed. This data forms the contributions C_1, C_2, and C_3.

To more easily analyse the results, performance graphs should be created using the method described above in *Artifact Execution*. This method of graph generation was used to create the graphs for the study where appropriate or interesting. Displaying the data graphically allows for much simpler interpretation of the results; at what problem size the GPU becomes worthwhile for use for each problem type (i.e. the *offload threshold*), and the effect the data transfer type has on achieved GEMM / GEMV performance. The former of these graph evaluations adds to contribution C_3, whilst the latter forms the contribution C_4.

Generally, a higher *offload threshold* (i.e. a larger problem size) equates to the CPU having a longer-lasting compute advantage over the GPU. This is often due to either a) data is moved to and from the GPU too often and prevents the compute advantage of the GPU being fully realised (common for Transfer-Always), or b) a given problem has a low *Arithmetic Intensity* and so it is beneficial to keep the problem resident on the CPU for larger and larger problem sizes. As the iteration count is increased, it is expected that the Transfer-Once and USM *offload thresholds* stays the same or decreases, whilst the Transfer-Always *offload threshold* increases.

Using the *offload thresholds* in combination with performance graph analysis is the recommended way of assessing when it becomes beneficial to offload a GEMM or GEMV problem to the GPU; for a given problem type and iteration count. This is also how the three different data transfer methods should be compared and how one should assess how data movement frequency can play a large role as to when offloading to the GPU is worthwhile.