Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Supplementary Material to DVFS-Aware Energy-Efficient Implementation of DNNs on Heterogeneous Hybrid Parallel Computers

HAMIDREZA KHALEGHZADEH[1], ATEFEH KHAZAEI[1], AND ALEXEY LASTOVETSKY[2](Member, IEEE)

[1]University of Portsmouth, School of Computing, Portsmouth PO1 3HE, United Kingdom (e-mail: hamidreza.khaleghzadeh@port.ac.uk; atefeh.khazaei@port.ac.uk)

[2]University College Dublin, School of Computer Science, Belfield, Dublin, Ireland (e-mail: alexey.lastovetsky@ucd.ie)

Corresponding author: Alexey Lastovetsky (e-mail: alexey.lastovetsky@ucd.ie).

This publication has emanated from research conducted with the financial support of TaighdeÉireann – Research Ireland under Grant number 20/FFP-P/8683. Additional financial support was provided by the Faculty of Technology, University of Portsmouth, under the Development Projects Funding (DPF) scheme 2024/25.

The supplementary material to the main manuscript, "DVFS-Aware Energy-Efficient Implementation of DNNs on Heterogeneous Hybrid Parallel Computers" includes:

- The implementation details of parallel training kernels utilised in the hybrid DNN application for CPU and accelerators.
- The specification of servers used for experimental evaluation.
- The methodology to configure the DVFS states on CPU and GPU cores.
- The scalability analysis of the HEPOPTA_Merge algorithm.

I. HYBRID DNN IMPLEMENTATION

This section presents a parallel and portable implementation of fully connected MLP DNN applications, designed for training on heterogeneous hybrid servers. We utilise this methodology to implement the DNN applications used in our experimental evaluation.

The implementation employs the Gradient Descent (GD) iterative algorithm, widely utilised for training DNN weights. To enable simultaneous parallel training across different types of computing devices, it is developed using a hybrid programming model featuring two levels of parallelism: (i) kernel-level and (ii) application-level.

The kernel-level parallelism utilises multi-threaded programming models to implement the parallel GD training algorithm (kernel) across various computing devices, including multicore CPUs and accelerators such as Nvidia GPUs.

Algorithm 1 illustrates the parallel kernel for multicore CPUs, implemented using OpenMP [1]. OpenMP is a standard API for parallel programming on multicore and shared-memory systems, employing directives such as `#pragma omp parallel for` to parallelise `for` loops by creating a team of threads and distributing loop iterations among them for concurrent execution. In Algorithm 1, line 4 parallelises the computations (forward pass and backpropagation) for each batch using the specified pragma.

The multi-threaded GD training kernels for accelerators are implemented using OpenACC [2], [3], another directive-based programming API that facilitates parallelisation and offloading of DNN training computations onto various accelerators, such as Nvidia GPUs. Since it is not limited to a specific computing device, OpenACC enables the development of portable parallel code that can execute across diverse hardware architectures without requiring source code modifications for each specific platform.

The multi-threaded kernel for training the neural network on accelerators is presented in Algorithm 2. In line 6, the OpenACC directive `#pragma acc parallel for` is employed to parallelise computations for each batch on a large number of accelerator cores.

To manage data transfer between host and accelerator main memories, Algorithm 2 utilises the OpenACC directive `#pragma acc data` with two clauses: `copyin` and

Algorithm 1: Implementation of Training Kernel for CPUs

```
1  Function Training_Kernel_CPU(batches, num_batches,
   batch_size, weights, biases);
   Input :
     • batches: A 1D array of training labelled samples.
     • num_batches: The number of batches in batches for training.
     • batch_size: The number of samples in each batch.
   In-Out :
     • weights: A 1D array of 2D matrices, where the i-th matrix
       (0 ≤ i < num_layers) contains a local copy of the network's weights for
       the i-th layer. This array is passed into the kernel by reference. Consequently, it
       holds the caller's weights when the training kernel is invoked, and any
       modifications made to this array within the kernel will directly affect the local
       weights array in the caller's scope.
     • biases: A 2D array, where the i-th column (0 ≤ i < num_layers) contains
       a local copy of the network's biases for the i-th layer. Similar to the weights
       array, this array is passed to the kernel by reference, ensuring that any
       modifications made to it within the kernel are reflected in the local bias array in
       the caller's scope.
2  begin
3      for b = 0; b < num_batches; b++ do
4          #pragma omp parallel for
5          for s = 0; s < batch_size; s++ do
               // Forward pass across input, hidden,
                  and output layers
               // to compute all activations
6              compute_forward_pass_CPU(batches[b * batch_size + s],
                 weights, biases);
               // Backpropagation and error calculation
7              calculate_errors_and_deltas_CPU(batches[b * batch_size +
                 s], weights, biases);
8          end
           // Sum hidden layer values and deltas
9          sum_values_and_deltas_CPU(···);
           // Update weights and biases
10         update_weights_and_biases_CPU(...);
11     end
12 end
```

Algorithm 2: Implementation of Training Kernel for Accelerators

```
1  Function Training_Kernel_ACC(batches, num_batches,
   batch_size, weights, biases);
   Input :
     • batches: A 1D array of training labelled samples.
     • num_batches: The number of batches in batches for training.
     • batch_size: The number of samples in each batch.
   In-Out :
     • weights: A 1D array of 2D matrices, where the i-th matrix
       (0 ≤ i < num_layers) contains a local copy of the network's weights for
       the i-th layer. It holds the caller's weights when the kernel is invoked, and upon
       kernel completion, the updated weights are copied back to the caller's local
       weights array.
     • biases: A 2D array, where the i-th column (0 ≤ i < num_layers) contains
       a local copy of the network's biases for the i-th layer. It holds the caller's biases
       when the training kernel is invoked, and upon kernel completion, the updated
       biases are copied back to the caller's local bias array.
2  begin
       // Data copy for accelerators
3      #pragma acc data copyin (batches, num_batches, batch_size) copy
         (weights, biases);
4      begin
5          for b = 0; b < num_batches; b++ do
6              #pragma acc parallel loop
7              for s = 0; s < batch_size; s++ do
                   // Forward pass across input, hidden,
                      and output layers
                   // to compute all activations
8                  compute_forward_pass_ACC(batches[b * batch_size +
                     s], weights, biases);
                   // Backpropagation and error
                      calculation
9                  calculate_errors_and_deltas_ACC(batches[b *
                     batch_size + s], weights, biases);
10             end
               // Sum hidden layer values and deltas
11             sum_values_and_deltas_ACC(···);
               // Update weights and biases
12             update_weights_and_biases_ACC(...);
13         end
14     end
15 end
```

`copy`, as shown in line 3. The `copyin` clause allocates memory on the accelerator and transfers data from the host to the accelerator, while the `copy` clause transfers the updated results (trained weights) back from the accelerator to the host memory. A comprehensive discussion of the kernel design using OpenMP and OpenACC is presented in [4].

At the application-level parallelism, as shown in Algorithm 3, CPU threads are created using Pthreads [5] to simultaneously invoke the designed parallel kernels across multiple devices on a heterogeneous platform, consisting of p devices: one multicore CPU and $p-1$ accelerators. This implementation integrates all necessary training kernels, including those for the CPU and accelerators, into a unified hybrid parallel DNN application to perform training computations using all computing devices of the heterogeneous platform in parallel.

Algorithm 3 invokes the `distribute_batches_dvfs` function in line 4. It determines the batch distribution and the DVFS settings for all the computing devices involved. The function returns three outputs:

- Batch distribution vector (X): It decides how to divide the entire set of training batches among the available p devices, as the first decision vector.
- DVFS setting vector (Θ): It determines the optimal DVFS state for each device, as the second decision vector.
- Created batches ($batches$): It creates the output vector $batches$, where `batches[i]` is a subset of `input_data` containing the batches allocated to the i-th device for training.

After setting the operating frequency of all p devices according to Θ (line 5), the training kernels are invoked to process their allocated workloads in parallel.

The main reasons for using Pthreads instead of OpenMP at this level of parallelism are to overcome certain compiler limitations, such as those of the Nvidia HPC C (NVC) compiler, which struggles with the compilation of nested parallelism in OpenMP code, and to provide better control over the mapping of CPU threads to cores compared with OpenMP.

II. EXPERIMENTAL PLATFORM SPECIFICATIONS

In this paper, we evaluates the efficiency of the proposed bi-objective optimisation approach on two heterogeneous platforms, HCLServer02 and HCLServer03. The first server

Algorithm 3: Hybrid DNN Training Application

Function `Hybrid_DNN_Training(input_data, batch_size, p, F, G, weights, biases);`
Input :
- *input_data*: A 1D array of training labelled samples.
- *batch_size*: The number of samples in each batch.
- *p*: The number of devices, including 1 CPU and p-1 accelerators, for parallel run.
- *F(x,Θ)*: Execution time profiles, as defined in the main manuscript.
- *G(x,Θ)*: Dynamic energy profiles, as defined in the main manuscript.

Output:
- *weights*: A 1D array of 2D matrices, where the i-th matrix ($0 \leq i < num_layers$) contains the trained network's weights for the i-th layer.
- *biases*: A 2D array, where the i-th column ($0 \leq i < num_layers$) contains the trained network's biases for the i-th layer.

begin
  `// Initialise weights and biases matrices`
  init_weights_biases(weights, biases);
  `// Create batches, distribute them between CPU and accelerators,`
  `// and determine processors' DVFS states`
  (X, Θ, batches) = distribute_batches_dvfs (input_data, batch_size, p, F, G);
  `// Set the operational frequency of each device`
  set_devices_frequencies(p, Θ);
  `// Loop for epochs`
  for $e = 0;\ e < num_epochs;\ e++$ **do**
    `// Create local copies of global weights and biases arrays`
    weights_local[0] = weights; biases_local[0] = biases;
    `// Execute CPU kernel on a multicore CPU`
    pthread_create(thread_id[0], Training_Kernel_CPU(batches[0], X[0], batch_size, weights_local[0], biases_local[0]));
    `// Execute accelerator kernels on` $p-1$ `devices`
    for $d = 1;\ d < p;\ d++$ **do**
      `// Create local copies of global weights and biases arrays`
      weights_local[d] = weights; biases_local[d] = biases;
      pthread_create(thread_id[d], Training_Kernel_ACC(batches[d], X[d], batch_size, weights_local[d], biases_local[d]));
    end
    `// Wait for all` p `threads to finish`
    for $d = 0;\ d < p;\ d++$ **do**
      pthread_join(thread_id[d]);
    end
    `// Average all local weights and biases to obtain global weights and biases`
    weights = average_local_weights(weights_local);
    biases = average_local_biases(biases_local);
    `// Calculate error using averaged weights and biases`
    calculate_error(...);
  end
  return $(weights, biases)$;
end

consists of an Intel Xeon Gold 6152 CPU connected to an NVIDIA P100 PCIe GPU, and the second features an Intel Xeon Platinum 8362 CPU paired with an NVIDIA A40 GPU. The specifications of these platforms are presented in Table 1.

III. CONFIGURING THE DVFS STATES

To incorporate DVFS states into the profiles, we need to measure the DNN application characteristics for the different DVFS states supported by each abstract processor.

A DVFS state is defined as a set of *(V, f)* pairs, where each pair specifies the operating voltage and frequency of a processor component with independent power control. The standard and reliable practice for modifying a DVFS state is to adjust the frequencies, since direct voltage manipulation may compromise system stability. Furthermore, application developers typically have access only to the frequency settings of DVFS states. Once a frequency is specified, the corresponding voltage is automatically set by the hardware, which maintains a fixed internal mapping between operating voltages and frequencies.

Unix-based kernels provide frequency governors that control how the CPU adjusts its operating frequency and voltage. Each governor employs specific policies to determine the real-time operating frequencies of CPU cores. Users with root privileges can select and configure these governors to suit application requirements. In this paper, we utilise two governors: (i) on-demand and (ii) userspace.

The on-demand DVFS governor dynamically adjusts the CPU frequency based on system load, increasing the frequency during high demand and lowering it when the system is less loaded. In contrast, the userspace governor allows user programs to explicitly set the processor's frequency, rather than relying on automatic kernel decisions.

To set the frequencies of Intel CPU cores, we employ `cpupower`, a Linux utility for controlling and monitoring CPU frequency scaling. For example, the following command sets the CPU frequency to a specified userspace value:

```
sudo cpupower frequency-set -g userspace
sudo cpupower frequency-set -d 2.0GHz
sudo cpupower frequency-set -u 2.0GHz
sudo cpupower frequency-set -f 2.0GHz
```

These commands first set the frequency scaling governor to `userspace` (`-g`), enabling manual frequency control by the user. Then, the minimum (`-d`), maximum (`-u`), and current frequency (`-f`) of all CPU cores are set to 2.0 GHz. This ensures that all CPU cores operate at a fixed frequency, avoiding any frequency variability.

To obtain the supported frequency range for the CPU cores, we employ the following Linux command:

```
sudo cpupower frequency-info
```

The following commands configure all Intel Xeon Platinum CPU cores to use the on-demand governor within the supported frequency range of 0.8 GHz to 3.6 GHz:

```
sudo cpupower frequency-set -g on-demand
sudo cpupower frequency-set -d 0.8GHz
sudo cpupower frequency-set -u 3.6GHz
```

Nvidia GPUs allow users to query the supported frequency ranges using the `nvidia-smi` tool as below:

```
nvidia-smi -q -d SUPPORTED_CLOCKS
```

We employ the `nvidia-smi -ac <MM_FREQ>, <SM_FREQ>` Linux command to set DVFS states for GPU memory and GPU SM cores. For instance, the following

TABLE 1: Specifications of Heterogeneous Hybrid Nodes: HCLServer02 and HCLServer03

Feature	HCLServer02	HCLServer03
CPU Specifications		
Processor	Intel Xeon Gold 6152	Intel Xeon Platinum 8362
Socket(s)	1	1
Cores per Socket	22	32
L1d Cache	32 KB	1,536 KB
L1i Cache	32 KB	1,024 KB
L2 Cache	256 KB	40,960 KB
L3 Cache	30,976 KB	49,152 KB
Total Main Memory	96 GB	64 GB
CPU TDP	140 W	265 W
Operating System	CentOS Linux 7	Ubuntu 22.10
GPU Specifications		
GPU Model	NVIDIA P100 PCIe	NVIDIA A40
Number of GPUs	1	2
CUDA Cores	3,584 (Pascal)	10,752 (Ampere)
Total Board Memory	12 GB CoWoS HBM2	48 GB GDDR6
Memory Bandwidth	549 GB/s	696 GB/s
GPU TDP	250 W	300 W

command sets the memory and core frequencies of the Nvidia A40 GPU to 7,251 MHz and 1,740 MHz, respectively.

```
nvidia-smi -ac 7251,1740
```

IV. SCALABILITY ANALYSIS OF HEPOPTA_MERGE

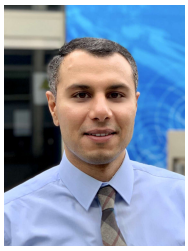
In this paper, we propose the HEPOPTA_Merge algorithm to obtain Pareto fronts for performance and energy consumption for different DVFS configurations. It first applies the HEPOPTA algorithm for each configuration, then merges the resulting fronts to construct the global Pareto front.

As proved in [6], HEPOPTA exhibits polynomial complexity. In addition, for any configuration $\Theta \in C_0 \times C_1 \times \cdots \times C_{p-1}$, the number of Pareto-optimal solutions returned by HEPOPTA does not exceed $m \times p$, where m denotes the maximum cardinality of the functions in F and G. Consequently, the total number of solutions that must be sorted and filtered to obtain the merged Pareto front is limited to $|C_0 \times C_1 \times \cdots \times C_{p-1}| \times m \times p$.

It is important to note that the number of DVFS states available for each processor (C_i, $i \in \{0, 1, \cdots, p-1\}$) is limited; therefore, $|C_0 \times C_1 \times \cdots \times C_{p-1}|$ is bounded by a constant in computing systems. Thus, we conclude that the HEPOPTA_Merge function has polynomial complexity and demonstrates scalability in realistic assumptions.

REFERENCES

[1] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," IEEE computational science and engineering, vol. 5, no. 1, pp. 46–55, 1998.
[2] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc—first experiences with real-world applications," in Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18. Springer, 2012, pp. 859–870.
[3] OpenACC, "OpenACC: Directives for Accelerators," http://www.openacc.org, 2024, [Online].
[4] H. Khaleghzadeh, A. Khazaei, and A. Lastovetsky, "Parallel DNNs on hybrid heterogeneous platforms: Design, implementation, and optimisation for performance and energy," IEEE Access, vol. 14, pp. 48 723–48 740, 2026.
[5] B. Nichols, D. Buttlar, and J. Farrell, Pthreads programming: A POSIX standard for better multiprocessing. " O'Reilly Media, Inc.", 1996.
[6] H. Khaleghzadeh, M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on heterogeneous hpc platforms for performance and energy through workload distribution," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, pp. 543–560, 2020.



HAMIDREZA KHALEGHZADEH obtained his B.Sc. and M.Sc. degrees in Computer Engineering (Software) in 2007 and 2011, respectively. He received a PhD from the School of Computer Science at University College Dublin, Ireland, in 2019. He is currently a Senior Lecturer in the School of Computing at the University of Portsmouth, UK. His research focuses on optimising performance and energy consumption in massively heterogeneous systems, high-performance heterogeneous systems, energy efficiency, and Big Data applications.



ATEFEH KHAZAEI obtained her Ph.D. from the School of Computer Science at University College Dublin (UCD) in 2025. She earned her B.Sc. in Computer Engineering (first rank) from Imam Reza International University, Mashhad, Iran, in 2009, followed by an M.Sc. in Software Engineering from Shiraz University, Shiraz, Iran, in 2011. Her research interests span data analysis, overlapping clustering, text mining, high-performance computing, and software security. She is currently affiliated with the School of Computing at the University of Portsmouth, UK.



ALEXEY LASTOVETSKY received the PhD degree from the Moscow Aviation Institute in 1986, and the Doctor of Sciences degree (Habilitation) from the Russian Academy of Sciences in 1997. His main research interests include heterogeneous parallel computing and energy-efficient parallel computing. He authored the monographs Parallel Computing on Heterogeneous Networks (Wiley, 2003) and High Performance Heterogeneous Computing (Wiley, 2009). He is an inaugural ScholarGPS Highly Ranked Scholar - Lifetime (ranked within 0.05% of all scholars in Parallel Computing).