Hindawi
International Journal of Reconfigurable Computing
Volume 2019, Article ID 3679839, 13 pages
https://doi.org/10.1155/2019/3679839

Research Article

Implementing and Evaluating an Heterogeneous, Scalable, Tridiagonal Linear System Solver with OpenCL to Target FPGAs, GPUs, and CPUs

Hamish J. Macintosh,[1,2] **Jasmine E. Banks**,[1] **and Neil A. Kelson**[2]

[1]*School of Electrical Engineering and Computer Science, Queensland University of Technology, Brisbane, Queensland 4001, Australia*
[2]*eResearch Office, Division of Research and Innovation, Queensland University of Technology, Brisbane, Queensland 4001, Australia*

Correspondence should be addressed to Hamish J. Macintosh; hj.macintosh@hdr.qut.edu.au

Received 27 February 2019; Revised 3 August 2019; Accepted 6 September 2019; Published 13 October 2019

Guest Editor: Sven-Bodo Scholz

Solving diagonally dominant tridiagonal linear systems is a common problem in scientific high-performance computing (HPC). Furthermore, it is becoming more commonplace for HPC platforms to utilise a heterogeneous combination of computing devices. Whilst it is desirable to design faster implementations of parallel linear system solvers, power consumption concerns are increasing in priority. This work presents the *oclspkt* routine. The *oclspkt* routine is a heterogeneous OpenCL implementation of the truncated SPIKE algorithm that can use FPGAs, GPUs, and CPUs to concurrently accelerate the solving of diagonally dominant tridiagonal linear systems. The routine is designed to solve tridiagonal systems of any size and can dynamically allocate optimised workloads to each accelerator in a heterogeneous environment depending on the accelerator's compute performance. The truncated SPIKE FPGA solver is developed first for optimising OpenCL device kernel performance, global memory bandwidth, and interleaved host to device memory transactions. The FPGA OpenCL kernel code is then refactored and optimised to best exploit the underlying architecture of the CPU and GPU. An optimised TDMA OpenCL kernel is also developed to act as a serial baseline performance comparison for the parallel truncated SPIKE kernel since no FPGA tridiagonal solver capable of solving large tridiagonal systems was available at the time of development. The individual GPU, CPU, and FPGA solvers of the *oclspkt* routine are 110%, 150%, and 170% faster, respectively, than comparable device-optimised third-party solvers and applicable baselines. Assessing heterogeneous combinations of compute devices, the GPU + FPGA combination is found to have the best compute performance and the FPGA-only configuration is found to have the best overall estimated energy efficiency.

1. Introduction

Given the ubiquity of tridiagonal linear system problems in engineering, economic, and scientific fields, it is no surprise that significant research has been undertaken to address the need for larger models and higher resolution simulations. Demand for solvers for massive linear systems that are faster and more memory efficient is ever increasing. First proposed in 1978 by Sameh and Kuck [1] and later refined in 2006 [2], the SPIKE algorithm is becoming an increasingly popular method for solving banded linear system problems [3–7]. The SPIKE algorithm has been shown to be an effective method for decomposing massive matrices whilst remaining numerically stable and demanding little memory overhead [8]. The SPIKE algorithm has been implemented with good results to solve banded linear systems using CPUs and GPUs and in CPU + GPU heterogeneous environments often using vendor-specific programming paradigms [6].

A scalable SPIKE implementation targeting CPUs and GPUs in a clustered HPC environment to solve massive diagonally dominant linear systems has previously been demonstrated with good computation and communication efficiency [5]. Whilst it is desirable to design faster implementations of parallel linear system solvers, it is necessary also to have regard for power consumption, since this is a primary barrier to exascale computing when using traditional general purpose CPU and GPU hardware [9, 10].

FPGA accelerator cards require an order of magnitude less power compared to HPC grade CPUs and GPUs. Previous efforts in developing FPGA-based routines to solve tridiagonal systems have been limited to solving small systems with the serial Thomas algorithm [11–13]. We have previously investigated the feasibility of FPGA implementations of parallel algorithms including the parallel cyclic reduction and SPIKE [14] for solving small tridiagonal linear systems. This previous work utilised OpenCL to produce portable implementations to target FPGAs and GPUs. The current work again utilises OpenCL since this programming framework allows developers to target a wide range of compute devices including FPGAs, CPUs, and GPUs with a unified language.

An OpenCL application consists of C-based kernel code intended to execute on a compute device and C/C++ host code that calls OpenCL API's to set up the environment and orchestrate memory transfers and kernel execution. In OpenCL's programming model, a device's computer resources are divided up at the smallest level as processing elements (PEs), and depending on the device architecture, one or more PEs are grouped into one or many compute units (CUs) [15]. Similarly, the threads of device kernel code are called work items (WIs) and are grouped into work groups (WGs). WIs and WGs are mapped to the PE and CU hardware, respectively.

OpenCL's memory model abstracts the types of memory that a device has available. These are defined by OpenCL as global, local, and private memory. Global memory is generally hi-capacity off-chip memory banks that can be accessed by all PEs across the device. Local memory is on-chip memory and has higher bandwidth and lower capacity than global memory and is only accessible to PE of the same CU. Finally, private memory refers to on-chip register memory space and is only accessible within a particular PE.

The motivation for this paper is to evaluate the feasibility of utilising FPGAs, along with GPUs and CPUs concurrently in a heterogeneous computing environment in order to accelerate the solving of a diagonally dominant tridiagonal linear system. In addition, we aimed at developing a solution that maintained portability whilst providing an optimised code base for each target device architecture and was capable of solving large systems. As such, we present the *oclspkt* routine, an heterogeneous OpenCL implementation of the truncated SPIKE algorithm that can dynamically load balance work allocated to FPGAs, GPUs, and CPUs concurrently or in isolation, in order to solve tridiagonal linear systems of any size. We evaluate the *oclspkt* routine in terms of computational characteristics, numerical accuracy, and estimated energy consumption.

This paper is structured as follows: Section 2 provides an introduction to diagonally dominant tridiagonal linear systems and the truncated SPIKE algorithm. Section 3 describes the implementation of the *oclspkt*-FPGA OpenCL host and kernel code and the optimisation process. This is followed by the porting and optimisation of the *oclspkt*-FPGA kernel and host code to the GPU and CPU devices as *oclspkt*-GPU and *oclspkt*-CPU. Section 3 concludes with discussion of the integration of the three solvers to produce the heterogeneous *oclspkt* solver. In Section 4, the individual solvers are compared to optimised third-party tridiagonal linear systems solvers. The three solvers are further compared in terms of estimated energy efficiency, performance, and numerical accuracy in addition to an evaluation of different heterogeneous combinations of the *oclspkt*. Finally, in Section 5, we draw conclusions from the results and discuss the implications for future work.

2. Background

2.1. Tridiagonal Linear Systems. A coefficient band matrix with a bandwidth of $\beta = 1$ in the linear system $Ax = y$ is considered tridiagonal:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & & & \\ a_{2,1} & a_{2,2} & a_{2,3} & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ & & & a_{n,n-1} & a_{n,n} \end{bmatrix}, \tag{1}$$

$$d = \min \frac{\left|A_{i,i}\right|}{\sum_{i\neq j}\left|A_{i,j}\right|}. \tag{2}$$

For nonsingular diagonally dominant systems where $d > 1$ in equation (2), a special form of nonpivoting Gaussian elimination called the Thomas algorithm [16] can perform LU decomposition in $\Theta(n)$ operations. The Thomas algorithm provides good performance when solving small tridiagonal linear systems; however, since this algorithm is intrinsically serial, it fails to scale well in highly parallel computing environments. More advanced, inherently parallel methods must be applied if the problem requires solving large systems. Many parallel algorithms exist for solving tridiagonal and block tridiagonal linear systems and are implemented in well-established numerical libraries [17–19].

2.2. The SPIKE Algorithm. The SPIKE algorithm [2] is a polyalgorithm that uses domain decomposition to partition a banded matrix into mutually independent subsystems which can be solved concurrently. Consider the tridiagonal linear system $AX = Y$ where A is $n \times n$ in size with only a single right-hand side vector Y. We can partition the system into p partitions of m elements, where $k = (1, 2, \ldots, p)$, to give a main diagonal partition A_k, off-diagonal partitions B_k and C_k, and the right-hand side partition Y_k:

$$A_k = \begin{bmatrix} a_{i,j} & a_{i,j+1} & & & \\ a_{i+1,j} & a_{i+1,j+1} & a_{i+1,j+2} & & \\ & \ddots & \ddots & \ddots & \\ & & a_{i+m-2,j+m-3} & a_{i+m-2,j+m-2} & a_{i+m-2,j+m-1} \\ & & & a_{i+m-1,j+m-2} & a_{i+m-1,j+m-1} \end{bmatrix},$$

$$[B_k, C_k, Y_k] = \begin{bmatrix} 0 & a_{mk+1,m(k-1)} & y_{mk} \\ \vdots & \vdots & \vdots \\ a_{m(k+1)-1,m(k+1)} & 0 & y_{m(j+1)} \end{bmatrix}. \tag{3}$$

The coefficient matrix partitions are factorised so $A = DS$ where D is the main diagonal block matrix and S is the SPIKE matrix as shown in the following equation:

$$DS = \begin{bmatrix} A_1 & & & & \\ & A_2 & & & \\ & \ddots & \ddots & \ddots & \\ & & & A_{p-1} & \\ & & & & A_p \end{bmatrix} \cdot \begin{bmatrix} I & V_1 & & & \\ W_2 & I & V_2 & & \\ & \ddots & \ddots & \ddots & \\ & & W_{p-1} & I & V_{p-1} \\ & & & W_p & I \end{bmatrix}. \tag{4}$$

where $V_k = (A_k)^{-1}B_k$ for $k = 1, \ldots, p-1$ and $W_k = (A_k)^{-1}B_k$ for $k = 2, \ldots, p$. By first solving $DF = Y$, the solution can be retrieved by solving $SX = F$. As $SX = F$ is the same size as the original system, solving for X can be simplified by first extracting a reduced system of the boundary elements between partitions to form $\widehat{S}\widehat{X} = \widehat{F}$ as shown in equation (5), where t and b denote the top- and bottommost elements of the partition:

$$\begin{bmatrix} 1 & 0 & V_1^t & 0 & & & & \\ 0 & 1 & V_1^b & 0 & & & & \\ 0 & W_2^t & 1 & 0 & & & & \\ 0 & W_2^b & 0 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & & \\ & & & 1 & 0 & V_{p-1}^t & 0 \\ & & & 0 & 1 & V_{p-1}^b & 0 \\ & & & 0 & W_p^t & 1 & 0 \\ & & & 0 & W_p^b & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X_1^t \\ X_1^b \\ X_2^t \\ X_2^b \\ \vdots \\ X_{p-1}^t \\ X_{p-1}^b \\ X_p^t \\ X_p^b \end{bmatrix} = \begin{bmatrix} F_1^t \\ F_1^b \\ F_2^t \\ F_2^b \\ \vdots \\ F_{p-1}^t \\ F_{p-1}^b \\ F_p^t \\ F_p^b \end{bmatrix}. \tag{5}$$

The reduced system $\widehat{S}$ is a sparse banded matrix of size $2p \times 2p$ and has a bandwidth of 2. Polizzi and Sameh [2] proposed strategies to handle solving the reduced system. The truncated SPIKE algorithm states for a diagonally dominant system where $d > 1$ (equation (2)) the reduced SPIKE partitions V_k^t and W_k^b can be set to zero [2]. This truncated reduced system takes the form of $p-1$ independent systems as shown in equation (6) which can be solved easily using direct methods:

$$\begin{bmatrix} 1 & V_k^b \\ W_{k+1}^t & 1 \end{bmatrix} \begin{bmatrix} X_k^b \\ X_{k+1}^t \end{bmatrix} = \begin{bmatrix} F_k^b \\ F_{k+1}^t \end{bmatrix}, \quad k = 1, \ldots, p-1. \tag{6}$$

With $\widehat{X}$ computed, the remaining values of X can be found with perfect parallelism using the following equation:

$$\begin{cases} A_1X_1 = F_1 - V_1^bX_2^t, & \\ A_kX_k = F_k - V_k^bX_{k+1}^t - W_k^tX_{k-1}^b, & k = 2, \ldots, p-1. \\ A_pX_p = F_p - W_p^tX_{p-1}^b, & \end{cases} \tag{7}$$

Mikkelsen and Manguoglu [20] conducted a detailed error analysis of the truncated SPIKE algorithm and showed that a reasonable approximation of the upper bound of the infinity norm is dependent on the degree of diagonal dominance, the partition size, and bandwidth of the matrix given by

$$|\widehat{x} - x|_\infty \approx d^{-m/\beta}. \tag{8}$$

3. Implementation

The general SPIKE algorithm consists of four steps: (1) partitioning the system, (2) factorising the partitions, (3) extracting and solving the reduced system, and (4) recovering the overall solution.

For diagonally dominant tridiagonal linear systems, the truncated SPIKE algorithm may be employed. This requires only the bottom SPIKE element v_{km}^b and top SPIKE element w_{km+1}^t in order to resolve the boundary unknown elements x_{km}^b and x_{km+1}^t [2]. This decouples the partition from the rest of the matrix and can be achieved by performing only the forward-sweep steps of LU factorisation, referred to as LU_{FS}, and forward-sweep steps of UL factorisation, referred to as UL_{FS}. LU_{FS} and UL_{FS} will be computed for partitions k and $k+1$ for $k = 1, 2, \ldots, p-1$.

The factorised right-hand side elements f_{km}^b and f_{km+1}^t and SPIKE elements w_{km+1}^t and v_{km}^b are used to form and solve the reduced system $\widehat{S}_k\widehat{X}_k = \widehat{F}_k$ using equation (6) to produce x_{km}^b and x_{km+1}^t. This algorithmic step is referred to as *RS*.

The remaining elements of the solution X_k can then be recovered with equation (7) via the back-sweep step of LU, referred to as LU_{BS}, and the back-sweep step UL factorisation, referred to as UL_{BS}, on the top and bottom half of the partitions k and $k+1$, respectively. We use the Thomas algorithm to compute the forward- and back-sweep factorisation steps giving the overall complexity of our truncated SPIKE algorithm as $\mathcal{O}(n)$. A high-level overview of the anatomy and execution flow of our *oclspkt* routine is shown in Figure 1. The *oclspkt* solver expects the size of the system n, the RHS vector Y, and the tridiagonal matrix split into vectors of its lower diagonal L, main diagonal D, and upper diagonal U as inputs. The solution vector X is returned.

In the following subsections, we describe the truncated SPIKE algorithm implementation for the FPGA (*oclspkt*-FPGA) using OpenCL and the development considerations to obtain optimised performance. As a part of this process, we design and implement an optimised TDMA OpenCL kernel to act as a serial baseline performance comparison for

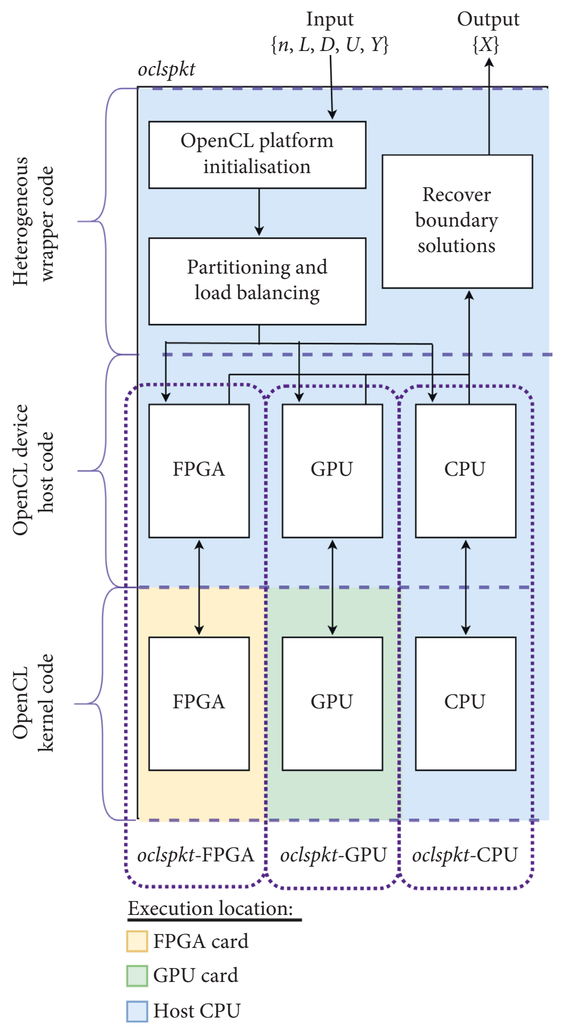

Figure 1: An overview of the anatomy and execution flow of the *oclspkt* solver.

the parallel truncated SPIKE kernel. Since the optimised TDMA implementation is constrained by the available global memory bandwidth, we are able to make genuine comparisons of FPGA hardware utilisation and computational complexity for these two kernels.

We then discuss the process of porting and optimising the *oclspkt* code for CPU and GPU. Finally, we describe integrating the three *oclspkt* (FPGA, GPU, and CPU) implementations as a heterogeneous solver. The specific hardware we target for these implementations are Bittware A10PL4 FPGA, NVIDIA M4000 GPU, and the Intel Xenon E5-1650 CPU.

3.1. FPGA Implementation. In order to take advantage of the FPGA's innate pipelined parallelism, we implement both the TDMA and truncated SPIKE algorithm as single work item kernels. A single work item kernel has no calls to the OpenCL API for the local or global relative position in a range of work items. This allows the Intel FPGA compiler to pipeline as much of the kernel code as possible, whilst not having to address WI execution synchronisation and access to shared memory resources. This reduces the OpenCL FPGA resource consumption overhead, allowing more of the logic fabric to be used for computation.

3.1.1. TDMA Kernel Code. We found no suitable FPGA implementation of a tridiagonal linear system solver able to solve large systems. In order to provide a suitable performance baseline for the more complex SPIKE algorithm, we implemented the Thomas algorithm or TDMA with OpenCL. The TDMA implementation calculates the forward-sweep and backsubsitution for one block of the input system at a time, effectively treating the FPGA's on-chip BRAM as cache for the current working data. The block size m is set as high as possible and is only limited by the available resources on the FPGA. An OpenCL representation of the kernel implementation is shown in Figure 2.

The forward-sweep section loads m elements of the input vectors L, D, U, Y, from off-chip DDR4 RAM to on-chip BRAM. With this input, the upper triangular and modified RHS is calculated, overwriting the initial values of D and Y. D and Y are then written back to DDR4 RAM due to BRAM limitation on the FPGA. The forward-sweep section iterates over $1, \dots, p$ blocks. The back-substitution section loads m elements of D, U, and Y vectors and writes m elements of X after recovering the solution via back-substitution. The back-substitution section iterates over $p, \dots, 1$ blocks.

3.1.2. Truncated SPIKE Kernel Code. An OpenCL algorithm representation of the truncated SPIKE kernel *spktrunc* is shown in Figure 3. The FPGA *oclspkt* implementation executes p iterations of its main loop, loading one block of the linear system given, where block size is m, and we solve 1 partition per iteration of the main loop.

A partition of size m is loaded from global memory and partitioned as per equation (3). $LU_{\mathrm{FS}}(k)$ and $UL_{\mathrm{FS}}(k)$ are executed concurrently to compute and store half of the upper and lower triangular systems $[D'UY']_k((m/2);\ m)$ and $[LD'Y']_k(1;\ (m/2))$, and the SPIKE elements v_k^b and w_k^t, respectively. Next, using y_{k-1}^b and v_{k-1}^b from the previous iteration and y_k^t and w_k^t as inputs for $RS(k)$, the boundary elements x_{k-1}^b and x_k^t are computed.

Finally, $X_k(1;\ (m/2))$ and $X_{k-1}((m/2);\ m)$ are then recovered with $UL_{\mathrm{BS}}(k)$ and $LU_{\mathrm{BS}}(k-1)$ of $[LD'Y']_k(1;\ (m/2))$ and $[D'UY']_{k-1}((m/2);\ m)$. $[D'UY']_k((m/2);\ m)$ and v_k^b are stored for the next iteration of the main loop. The FPGA solver is initialised with an upper triangular identity matrix in $[D'UY']_{k-1}((m/2);\ m)$ for $k = 0$.

This results in a streaming linear system solver where loading in a block of partitions at the start of the pipeline will compute a block of the solution vector with a $-(m/2)$ element offset.

3.1.3. Host Code. On the host side, in order to interleave the PCIe memory transfers to the device with the execution of the solver kernel, we create in-order command queues for writing to, executing on, and reading from the FPGA.

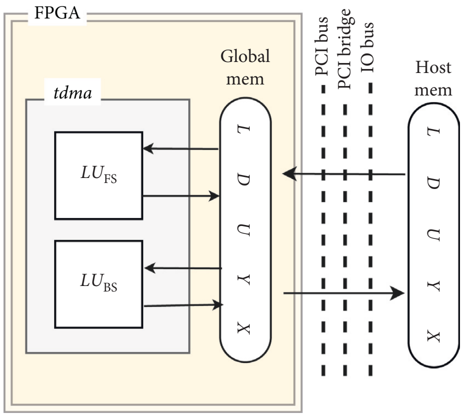

Figure 2: The FPGA TDMA OpenCL kernel *tdma*, with the execution path and data dependencies shown. The *tdma* executes as a single WI kernel.

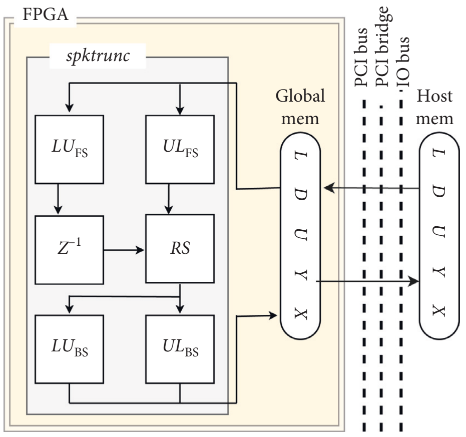

Figure 3: The FPGA truncated SPIKE OpenCL kernel *spktrunc*, with the execution path and data dependencies shown. The *spktrunc* executes as a single WI kernel.

We create two copies of read-only memory objects L, D, U, Y and a write-only memory object X. The *spktrunc* kernel and the FPGA's DMA controller for the PCIe bus share the total bandwidth of the DDR4 RAM bank. To maximise FPGA global memory bandwidth, the device memory objects are explicitly designated specific RAM bank locations on the FPGA card in such a way that the PCIe to device RAM, and device RAM to FPGA bandwidth, is optimised.

The execution kernel is enqueued as a 1-by-1-by-1 dimension task with arguments p, L, D, U, Y, and X, where p, the number of partitions to solve, is given by $\text{ceil}((n/m) + 1)$. The execution kernel is scheduled and synchronised with the write and read operations of the device memory objects using OpenCL event objects.

The kernel code is dependent on the partition size m, so memory buffers for the input and output vectors are created as 1-by-size vectors, where size is given by $p \times m$. The input matrix consists of lower, main, and upper diagonal vectors of A and a single right-hand side vector Y is stored in row-major order.

The memory objects L, D, U, Y are padded with an identity matrix and zeros in order to accommodate linear systems where m is not a factor of n, giving the overall memory requirement as $5 \times$ size.

As the kernel is implemented as a single work item, this allows for single-strided memory access patterns when the FPGA loads partitions from global memory for processing. This means that it is not necessary to implement a preexecution data marshalling stage as is often required for SIMD or SIMD-like processors.

3.1.4. Kernel Complexity and Hardware Utilisation. The FLOP requirements for our TDMA and truncated SPIKE OpenCL kernels are presented in Table 1. The TDMA kernel has significantly fewer FLOPs compared to the truncated SPIKE kernel. This is expected as the TDMA kernel only computes the LU factorisation and back-substitution, compared to the more computationally complex truncated SPIKE polyalgorithm described previously. However, since the TDMA kernel requires the upper triangular matrix of the entire system to be stored to global memory as an intermediate step and then subsequently reread, the TDMA kernel requires double the number of FPGA to off-chip memory transactions in comparison with the truncated SPIKE kernel.

The FLOP and memory transaction requirements shown in Table 1 are reflected in the FPGA kernel hardware utilisation presented in Table 2. OpenCL requires a static partition of the available FPGA hardware resources in order to facilitate host to FPGA memory transfers and OpenCL kernel execution. Per Table 2, this static partition is significant, consuming at least 10% of measured resource types. The total resource utilisation for each kernel is given by the addition of OpenCL static resource utilisation and the kernel-specific resource utilisation.

The more computationally complex truncated SPIKE kernel requires more lookup tables (ALUT), flip-flops (FF), and digital signal processor (DSP) tiles than the TDMA kernel. The TDMA kernel however requires more block RAM (BRAM) tiles due to implementing a greater number of load store units to cater for the extra global memory transactions. Furthermore, both kernels are constrained by the available amount of BRAM on the FPGA, with the BRAM utilisation by far the highest resource utilisation for both kernels.

3.1.5. FPGA OpenCL Optimisation Considerations. Our implementation of the truncated SPIKE algorithm is global memory bandwidth constrained. It requires large blocks of floating point data to be accessible at each stage of the algorithm. By far the largest bottleneck to computational throughput is ensuring coalesced aligned global memory transactions.

When loading matrix partitions from the global to local or private memory, a major optimisation consideration is the available bandwidth on the global memory bus. The available bandwidth per memory transaction is 512 bits, and the load-store units that are implemented by the Intel FPGA

Table 1: FLOP and global memory transactions required for the TDMA and truncated SPIKE FPGA kernels.

Operation	TDMA	Truncated SPIKE
ADD/SUB	$3mp$	$(5m+3)p$
MUL	$3mp$	$(6m+5)p$
DIV	$3mp$	$(3m+3)p$
MEM	$10mp$	$5mp$

Table 2: FPGA hardware utilisation for the OpenCL static portion, TDMA, and truncated SPIKE kernels, and total utilisation for each implementation (static + kernel).

Resource	OpenCL static (%)	TDMA		Truncated SPIKE	
		Kernel (%)	Total (%)	Kernel (%)	Total (%)
ALUTs	13	14	**27**	18	**31**
FFs	13	10	**23**	13	**23**
BRAMs	16	52	**68**	49	**65**
DSPs	10	16	**26**	27	**37**

compiler are of the size 2^b bits where $b_{\min} = 9$ and $b_{\max}$ is constrained by the available resources on the FPGA. Therefore, to ensure maximum global memory bandwidth with the aforementioned constraints, we set partition size m to 32 for single precision floating point data for both kernels resulting in 1024 bit memory transactions.

The value of m is hard-coded and known at compile time allowing for unrolling of the nested loops at the expense of increased hardware utilisation. Unrolling a loop effectively tells the compiler to generate a hardware instance for each iteration of the loop, meaning that if there are no loop-carried dependencies, the entire loop is executed in parallel. However, for loops with carried dependencies such as *LU*/*UL* factorisation, each iteration cannot execute in parallel. Nonetheless, this is still many times faster than sequential loop execution despite the increase in latency that is dependent on the loop size.

Loop unrolling is our primary computational optimisation step, thereby allowing enough compute bandwidth for our kernel to act as a streaming linear system solver. Note that in our optimisation process, we either fully unroll loops or not at all. It is possible to partially unroll loops for a performance boost when hardware utilisation limitations do not permit a full unroll. Partially unrolling a loop can however be inefficient since the hardware utilisation does not scale proportionally with the unroll factor due to the hardware overhead required to control the loop execution.

3.2. Porting the Truncated SPIKE Kernel to CPU and GPU. In order to investigate the full potential for a heterogeneous computing implementation of the truncated SPIKE algorithm for solving tridiagonal linear systems of any size, we exploited the portability of OpenCL. We modified the host and kernel code used for the FPGA implementation to target CPU and GPU hardware. To achieve this, it was necessary to make modification to the host and kernel side memory objects and data access patterns, remap the truncated SPIKE algorithm to different kernel objects, and modify the work group sizes and their mapping to compute units with respect to CPU and GPU hardware architecture.

3.2.1. Partitioning and Memory Mapping. The memory requirements for CPU and GPU implementations are dependent on the partitioning scheme for each device. The memory required to solve for a partition of size m multiplied by a multiple of the GPU's preferred work group size must be less than the available local memory. This constrains the size and number of partitions m, since it is preferable to maximise the occupancy of the SIMD lane whilst ensuring that sufficient local memory is available.

Unlike the GPU, all OpenCL memory objects on the CPU are automatically cached into local memory by hardware [21]. However, considering that the CPU has a lower compute unit count, we maximise the partition size m to minimise the number of partitions, thereby minimising the operation count required to recover the reduced system. The relative values for p and size in terms of m and work group size are shown in Table 3.

For our implementation of the truncated SPIKE algorithm for the CPU and GPU, the host and kernel memory requirements are five 1-by-size vectors, L, D, U, Y, X, of the partitioned system and four 1-by-$(p+2)$ vectors, V, W, Y^t, Y^b, of the reduced system. By storing the values for V, W, Y^t, Y^b in a separate global memory space, we remove the potential for bank conflicts in memory transactions that may occur if the reduced system vectors are stored in place in the partitioned system.

The reduced system memory objects are padded with zeros to accommodate the top- and bottommost partitions $j = 0$ and $j = p$ removing the need for excess control code in the kernel to manage the topmost and bottommost partitions. Further, to ensure data locality for coalesced memory transactions on both the CPU and GPU, the input matrix is transformed in a preexecution data marshalling step. The data marshalling transforms the input vectors so that data for adjacent work items are sequential instead of strided. This allows the data to be automatically cached and for

Table 3: *oclspkt* kernel partitioning schemes.

Device	Partitions (p)	size
FPGA	$\text{ceil}(n/m)+1$	$p\times m$
GPU	$n/(WG_{\text{size}}\times m)$	$p\times m\times WG_{\text{size}}$
CPU	$n/(CUs\times WG_{\text{size}})$	$p\times CU\times WG_{\text{size}}$

vector processing of work items on the CPU and for full bandwidth global memory transactions on the GPU.

3.2.2. Remapping the Kernel. In contrast to the FPGA implementation of the truncated SPIKE algorithm, for the CPU and GPU implementations, we split the code into two separate kernels for the CPU and three separate kernels for the GPU. This allows for better work group scheduling and dispatching for multiple compute unit architectures as we enqueue the kernels for execution as arrays of work items known as an NDRange in OpenCL's parlance.

In remapping the kernel to the CPU, the underlying architecture provides relatively few processing elements per compute unit and a fast clock speed. As such, in order to make best use of this architecture, the number of partitions of the truncated SPIKE algorithm should be minimised, thereby ensuring allocation of work groups of the maximum possible size to each compute unit to ensure maximum occupancy. Figure 4 shows an OpenCL representation of the CPU implementation, its execution order, and data path.

For the CPU implementation, we use the partitioning scheme proposed by Mendiratta [22]. We compute the LU and UL forward-sweep factorisation in the $spkfac_{cpu}$ kernel where we apply UL factorisation to elements 0 to $2m_{\text{min}}$ and apply LU factorisation to elements m_{min} to m where m_{min} is the smallest partition size required to purify the resulting factorisations of error as per equation (8). This reduces the overall operation count and is only possible when $m \gg m_{\text{min}}$. The $spktfac_{cpu}$ kernel is enqueued as a p-by-1-by-1 NDRange, in a single in-order command queue.

The reduced system and the recovery of the overall solution are handled by a second kernel *spktrec*. The *spktrec* kernel is enqueued as per $spktfac_{cpu}$ in a single in-order command queue. The reduced system is solved, and the boundary unknown elements are recovered and used to compute the UL back-sweep of elements 0 to $m_{\text{min}}-1$ and the LU back-sweep of elements m to m_{min}.

In contrast to the CPU, the GPU has many processing elements per compute unit and a relatively low clock speed. In order to optimise performance, it was important to maximise the number of partitions of the SPIKE algorithm, by reducing the partition size and thereby ensuring maximum occupancy of the processing elements. Figure 5 shows an OpenCL representation of the GPU implementation, its execution order, and data path.

For the GPU, partitioning the system and the LU and UL factorisation of the code are handled by the first kernel, $spkfact_{gpu}$. Unlike the CPU, the GPU computes the entire block size m of the UL and the LU factorisations. Only the top half of the UL and the bottom half of the LU results are then stored in global memory in order to reduce global

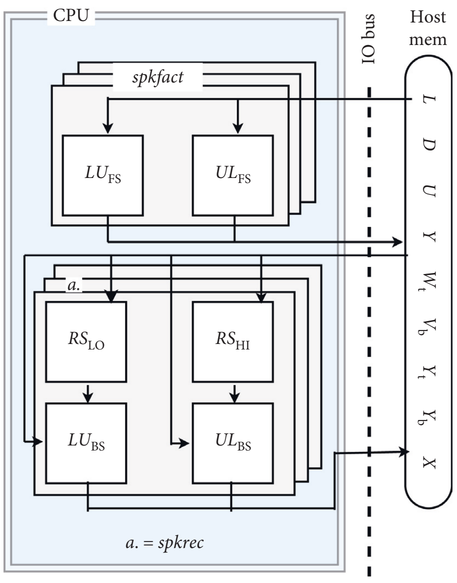

Figure 4: The CPU truncated SPIKE OpenCL kernels *spkfact* and *spkrec*, with the execution path and data dependencies shown. Both kernels are executed as an NDRange of work items.

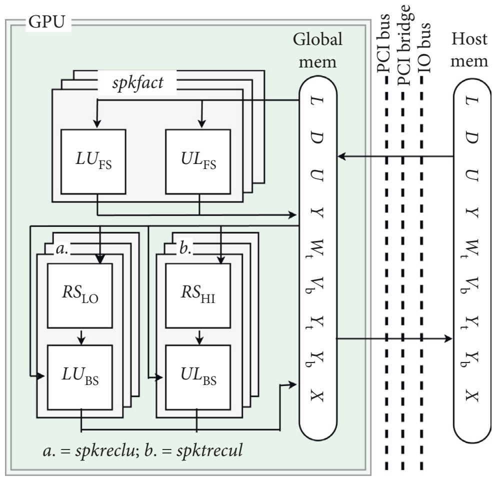

Figure 5: The GPU truncated SPIKE OpenCL kernels *spkfact*, *spkrecul*, and *spkreclu*, with the execution path and data dependencies shown. All kernels are executed as an NDRange of work items.

memory transactions and overall global memory space requirements. The reduced system and the recovery of the overall solution are handled by two kernels, *spktreclu* and *spktrecul*. *spktreclu* and *spktrecul* only load the bottom half of partition m and top half of partition m, respectively, to compute the back-sweep portions of LU and UL factorisation. The three kernels are again enqueued as p-by-1-by-1 NDRange in in-order command queues for writing to, executing on, and reading from the GPU. As with the FPGA, this effectively interleaves the PCIe data transfer with kernel execution.

3.3. The Heterogeneous Solver. We further extend our truncated SPIKE implementation to utilise all available computation resources available on a platform, as shown in Figure 1.

This heterogeneous solver first checks for available devices on the host using OpenCL APIs and then queries if device profiling data exist for found devices. If profiling data are not available for all devices, each device will be allocated an even portion of the input system and profiling data will be collected on the next execution of the solver. Otherwise, each device will be allocated a portion of the input system determined by the percentage of the total system throughput over the individual devices' previously recorded throughput. Throughput in this case includes data transit time across the PCIe bus, data marshalling, and compute time of the kernel.

The heterogeneous solver then asynchronously dispatches chunks of the input data to the devices, executes the device solvers, and recovers the solution. The interchunk boundary solutions recovered from the devices are cleansed of error by executing a "top" level of the truncated SPIKE algorithm on the chunk partitions.

4. Evaluation

In the following subsections, we evaluate the *oclspkt* routine in terms of compute performance, numerical stability, and estimated power efficiency. The results presented use single precision floating point and all matrices are random and nonsingular, and the main diagonal has a diagonal dominance factor $d > 3$.

All results presented in this paper have been executed on a Dell T5000 desktop PC with an Intel Xeon CPU E5-1620 v4, 64 GB of RAM, a Bittware A10PL4 FPGA, and a NVIDIA M4000 GPU; full specifications are listed in Table 4.

Table 4: Specifications for Dell T5000 desktop PC.

Component	Specification
CPU	Intel Xeon E5-1620 v4 @ 3.50 GHz
GPU	NVIDIA M4000 8 GB GDDR5 PCIe G3 x16
FPGA	Bittware A10PL4 w/ Intel Arria 10 GX 8 GB DDR4 PCIe G3 x8
RAM	64 GB DDR4 @ 2400 MHz
OS	CentOS 7.4
Software	ICC 18.0.3 CUDA 9.0 Intel Quartus Pro 17.0 Intel OpenCL SDK 7.0.0.2568

4.1. Compute Performance. To evaluate the compute performance of the *oclspkt*, we first only consider the kernel execution time for our target devices in isolation, assuming predistributed memory. In Figure 6, we show the time to solve a system where $N = 256 \times 10^6$, specifically identifying the solution and data marshalling components of the overall execution time. We set N to 256×10^6 in-order to showcase the best possible performance for the computing-device only without introducing PCI memory transactions. In this experiment, the GPU kernel takes on average 78.4 ms to solve the tridiagonal system, where the FPGA and CPU are 2.6 and 4.8× slower at 200 ms and 376 ms, respectively. Furthermore, when also considering the data marshalling overheads required by the GPU and CPU kernels, the GPU is still the quickest at 152 ms with the FPGA and CPU now 1.3 and 6.1× slower.

The compute performance figures are not surprising when we consider that the performance of oclspkt is bound by the available memory bandwidth. For large matrices, the global memory transactions required for oclspkt-GPU and oclspkt-CPU are $\approx 13N$ where the oclspkt-FPGA solver requires $\approx 5N$. We can estimate the performance of the compute devices using the required memory transactions of the individual solvers and with the total available memory bandwidth of the devices. Performance estimation is calculated using MT/B, where MT is the number of required memory transactions and B is the maximum available memory bandwidth. The estimated relative estimated performance is calculated to be 1, 2.17, and 4.57× slower for the GPU, FPGA, and CPU, respectively (normalised for the GPU). These values closely correspond to the measured relative performance of the kernel solve time in Figure 6.

In Figure 7, we compare these results to other diagonally dominant tridiagonal solver algorithms, our TDMA FPGA kernel, a CUDA-GPU implementation, *dgtsv*, [6], the Intel MKL *sdtsvb* routine [23], and a sequential CPU implementation of the TDMA. For each of the three target devices, our *oclspkt* implementation outperforms the comparison routines for solving a tridiagonal system of $N = 256 \times 10^6$. The *oclspkt* (FPGA) is 1.7× faster than the TDMA (FPGA) kernel, the *oclspkt* (GPU) implementation is 1.1× faster than the *dgtsv*, and our *oclspkt* (CPU) is 1.5 and 3.5× faster than the *sdtsvb* and TDMA CPU solvers, respectively. Note that for each of these results, we include any data marshalling overhead, but exclude host to PCIe device transfer time.

A comparison of the compute performance targeting single and heterogeneous combinations of devices executing the *oclspkt* routine is shown in Figure 8. We normalise the performance metric to rows solved per second (RSs^{-1}) to provide a fair algorithmic comparison across different device hardware architectures. Furthermore, when evaluating the heterogeneous solver performance of *oclspkt*, we use a holistic system approach, which includes the host to device PCIe data transfer times for the FPGA and GPU devices, and all data marshalling overheads. As shown in Table 5, the GPU + FPGA device combination has the best average maximum performance. The GPU + FPGA device combination performs 1.38× better than the next best device, the GPU-only, and performs 2.48× better than the worst performing device, the CPU-only implementation.

Curiously, we would expect performance metrics of the heterogeneous combinations of devices to be close to the summation of the individual device performance metrics. In fact, our results show that only the GPU + FPGA heterogeneous performance is close to the summation of the GPU and FPGA-only performance at 88% of the theoretical total. The CPU + FPGA, CPU + GPU, and CPU + GPU + FPGA average maximum performance are only 65%, 51%, and 55%, respectively.

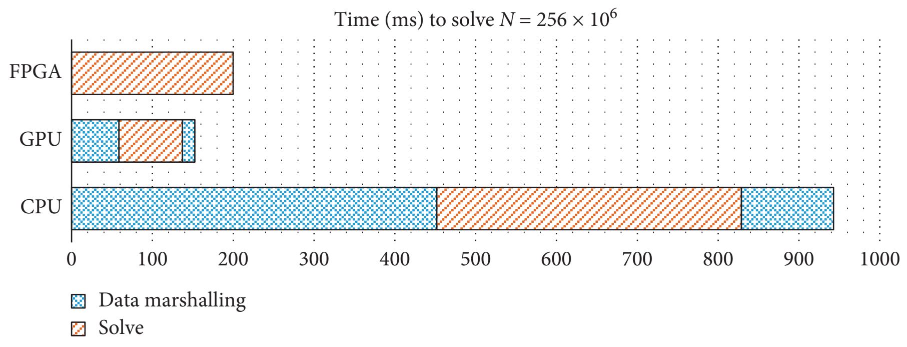

Figure 6: Time (ms) to solve a system of size $N = 256 \times 10^6$ using *oclspkt*, targeting CPU, GPU, and FPGA devices.

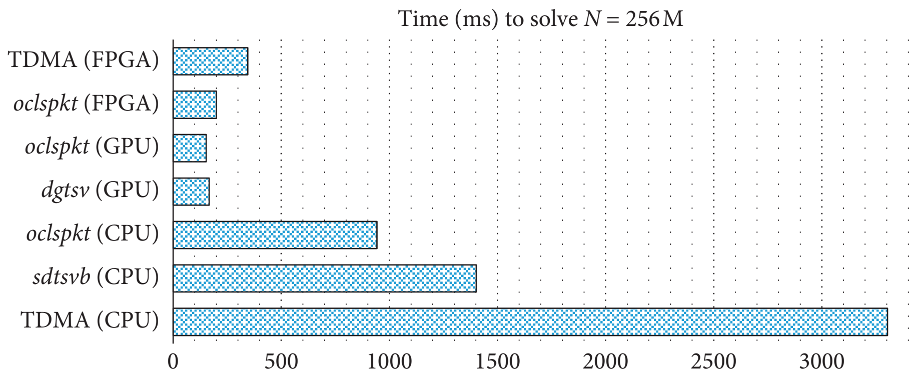

Figure 7: Comparing time (ms) to solve a system of size $N = 256 \times 10^6$ using *oclspkt*, *dgtsv*, *sdtsvb*, and TDMA.

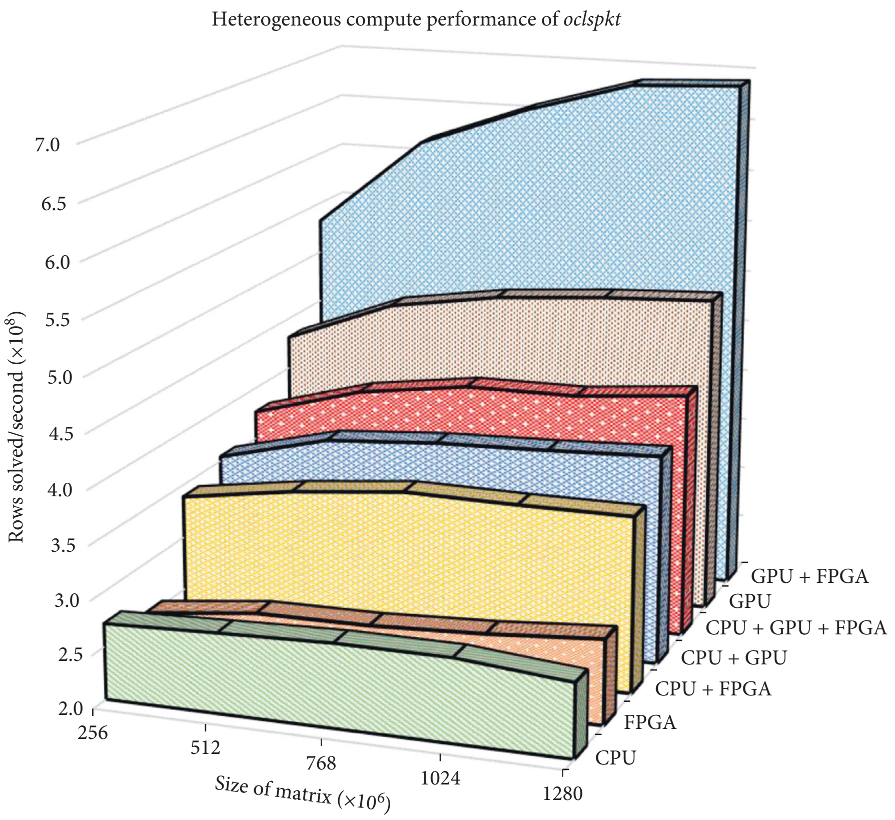

Figure 8: Performance comparison in rows solved per second for $N = (256 \dots 1280) \times 10^6$ when targeting CPU, GPU, FPGA, and heterogeneous combinations of devices.

Table 5: Maximum observed average ($n = 16$) compute performance and estimated energy efficiency of *oclspkt* for devices and heterogeneous combinations.

Device	Compute performance	Estimated energy efficiency
CPU	279	1.39
GPU	501	12.9
FPGA	280	**28.6**
CPU + GPU	396	1.67
CPU + FPGA	365	1.81
GPU + FPGA	**691**	15.6
CPU + GPU + FPGA	431	2.06

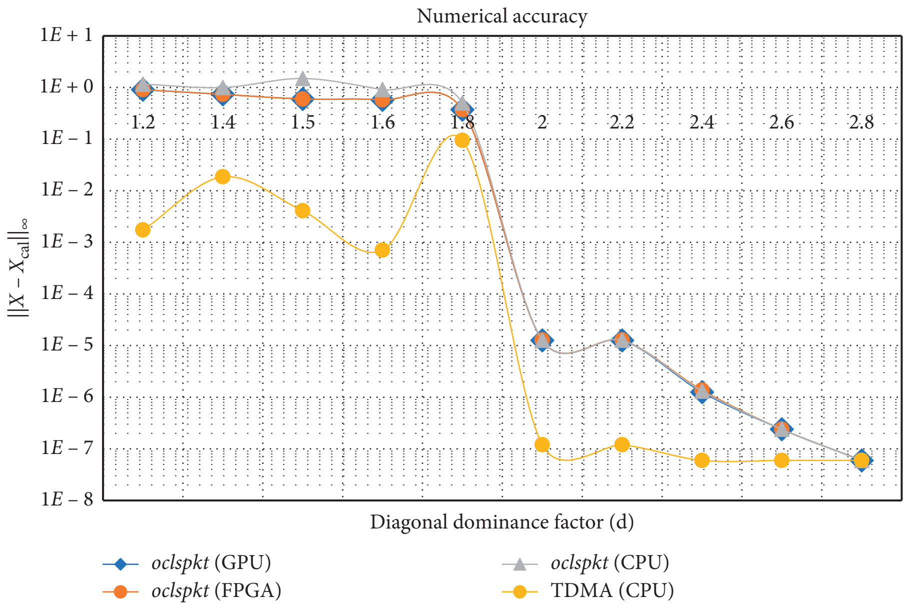

Figure 9: Numerical accuracy of *oclspkt* for varying diagonal dominance compared to CPU TDMA solver.

For the PCIe attached devices, the GPU and FPGA performance metrics are determined by the available PCIe bus bandwidth. We have provisioned primary and secondary memory spaces in the GPU and FPGA global memory space. This allows the *oclspkt* routine to execute the OpenCL kernel on the primary memory space whilst writing the next input chuck to the secondary memory space. This ensures the PCIe bus, and available memory bandwidth is being used in an efficient way. That is to say, the OpenCL kernel execution time (Figure 6) is completely interleaved with the host to device memory transfers. As such, the M4000 GPU card with 16 PCIe Gen 3.0 lanes available will outperform the A10PL4 FPGA card with 8 PCIe Gen 3.0 lanes regardless of the kernel compute performance. Similarly, the CPU performance is determined by the available host RAM bandwidth.

Using the de facto industry standard benchmark for measuring sustained memory bandwidth, STREAM [24, 25], our desktop machine, specified in Table 4, has a maximum measured memory bandwidth of 42 GBs^{-1}. Profiling our CPU implementation of *oclspkt* using Intel VTune Amplifier shows very efficient use of the available memory bandwidth with sustained average 36 GBs^{-1} and peak 39 GBs^{-1} memory bandwidth utilisation. This saturation of host memory bandwidth by the CPU solver creates a processing bottleneck and negatively affects the PCIe data transfer to the FPGA and GPU. This, coupled with the heterogeneous partitioning scheme described in subsection 3.3, will favour the increase in the chunk size of the input system allocated for CPU computation on each successive invocation of the *oclspkt* routine and subsequently decrease the performance of GPU and FPGA devices.

One thing that may increase the performance of the CPU + [GPU | FPGA | GPU + FPGA] combinations of solvers is to change the workload partitioning scheme to only look at data marshalling and kernel execution times, excluding the PCIe data transfer times. This would mean on successive calls of the routine the host would be tuned to send more workload to the GPU and FPGA and minimise the workload allocated to the CPU, thus improving performance.

4.2. Numerical Accuracy. In Figure 9, we show the numerical accuracy of the *oclspkt* in terms of the infinity norm of the known and calculated results, varied by the diagonal dominance of the input matrix compared to the TDMA CPU implementation. The TDMA approaches the machine

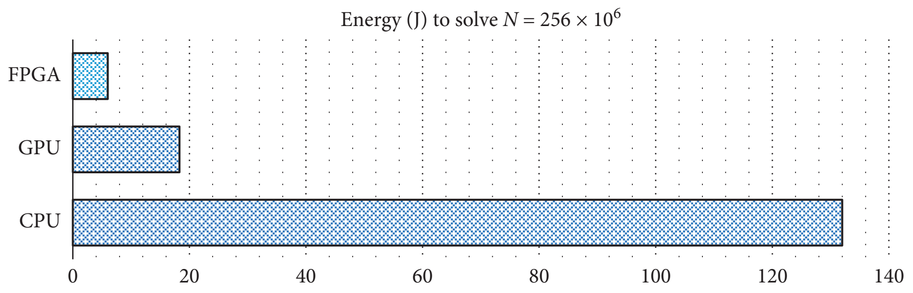

Figure 10: Joules required to solve a tridiagonal system of $N = 256 \times 10^6$ per device using the *oclspkt* routine.

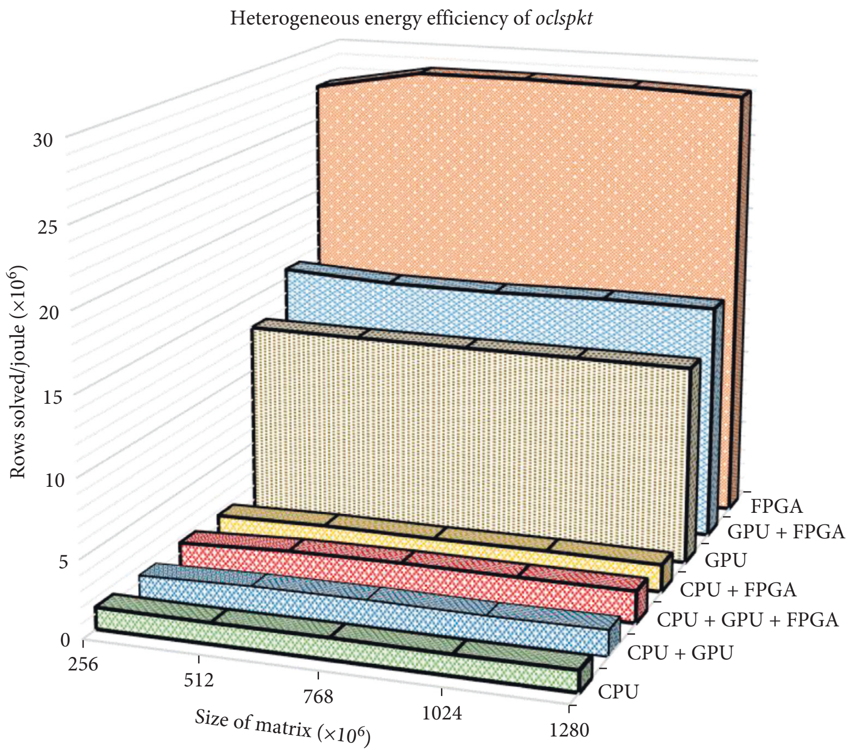

Figure 11: Estimated energy efficiency: comparison in rows solved per second for $N = (256 \ldots 1280) \times 10^6$ for *oclspkt* when targeting CPU, GPU, FPGA, and heterogeneous combinations of devices.

epsilon value for single precision floating point numbers when the diagonal dominance of the input system is 2, whereas the *oclspkt* for the CPU, GPU, and FPGA requires a diagonal dominance of 2.8 to achieve a similar accuracy. Equation (8) shows that an approximation of the upper bound of the infinity norm error is dependent on the SPIKE partition size, the bandwidth, and the degree of diagonal dominance. As the GPU and FPGA partition sizes and the small CPU partition sizes are equal, that is, $m_{\mathrm{GPU}} = m_{\mathrm{FPGA}} = m_{\mathrm{CPU}_{\min}}$, the numerical accuracy for all implementations is expected to be very similar.

4.3. Estimated Energy Consumption and Efficiency. To determine estimated energy consumption in Joules for each device, we used the manufacturer's rated thermal design power (TDP) and multiplied it by the kernel execution time for data marshalling and solved steps of the *oclspkt*. TDP represents the average power in watts used by a processor when the device is fully utilised. Whilst this is not a precise measurement of power used to solve the workload, it nevertheless provides a relative interdevice benchmark. The TDP of the M4000 GPU is 120 W, the Xeon E-1650 is 140 W, and the A10PL4 FPGA is 33 W.

When solving a system of $N = 256 \times 10^6$ as shown in Figure 10, the FPGA implementation uses 2.8× less and 20× less energy than the GPU and CPU implementations, respectively. Further, in Figure 11, we can see the estimated energy efficiency of each hardware configuration of *oclspkt* in rows solved per Joule. Across the range of the experiment, each solver shows consistent results with the FPGA-only solver estimated to be the most energy-efficient peaking at 28×10^6 rows solved per Joule.

The FPGA-only solver is estimated to be on average 1.8× more energy efficient than the next best-performing solver, the GPU + FPGA, and is 20.0× more energy efficient than the poorest performing CPU-only solver. This is not surprising since the TDP for the FPGA is an order of magnitude smaller than the other devices. Similarly to the heterogeneous results in subsection 4.1, the addition of the CPU solver significantly constrains the available bandwidth to host memory slowing down the PCIe data transfer rates. In turn, this pushes more work to the CPU solver and slows down the overall compute, and since the CPU has the highest TDP, this exacerbates the poor energy efficiency.

5. Conclusion

In this paper, we presented a numerically stable heterogeneous OpenCL implementation of the truncated SPIKE algorithm targeting FPGAs, GPUs, CPUs, and combinations of these devices. Our experimental case has demonstrated the feasibility of utilising FPGAs, along with GPUs and CPUs concurrently in a heterogeneous computing environment in order to accelerate the solving of a diagonally dominant tridiagonal linear system. When comparing our CPU, GPU, and FPGA implementation of *oclspkt* to a suitable baseline implementation and third-party solvers specifically designed and optimised for these devices, the compute performance of our implementation showed 150%, 110%, and 170% improvement, respectively.

Profiling the heterogeneous combinations of *oclspkt* showed that targeting the GPU + FPGA devices gives the best compute performance and targeting FPGA-only will give the best estimated energy efficiency. Also adding our highly optimised CPU implementation to a heterogeneous device combination with PCIe attached devices significantly reduced the expected performance of the overall system. In our experimental case, with a compute environment that has CPUs, GPUs, and FPGAs, it is advantageous to relegate the CPU to a purely task orchestration role instead of computation.

Under our experimental conditions, all device compute performance results are memory bandwidth constrained. Whilst the GPU kernel compute performance is several times faster than the FPGA kernel, the FPGA test hardware has several times less available memory bandwidth. As high-bandwidth-data transfer technology is introduced to the new generations of FPGA accelerator boards, this performance gap between devices is expected to close. Given the significantly lower power requirements, incorporation of FPGAs has the potential to reduce some of the power consumption barriers currently faced by HPC environments as we move towards exascale computing.

A natural progression of this work would be to extend the *oclspkt* routine to be able to solve nondiagonally dominant and block tridiagonal linear systems. Further, it would be advantageous to extend the heterogeneous partitioning routine to be able to tune the solver to maximise energy efficiency where desired. A part of this extension would involve a more detailed power analysis and direct in-line monitoring of the host power usage.

An extension of this work may also seek to account for memory bottlenecks detected on successive invocations of the solver further enhancing performance in heterogeneous applications.

Data Availability

The source code and data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This project utilised the High-Performance Computing (HPC) Facility at the Queensland University of Technology (QUT). The facility is administered by QUT's eResearch Department. Special thanks go to the eResearch Department for their support, particularly for providing access to specialist FPGA and GPU resources.

References

[1] A. H. Sameh and D. J. Kuck, "On stable parallel linear system solvers," *Journal of the ACM*, vol. 25, no. 1, pp. 81–91, 1978.

[2] E. Polizzi and A. H. Sameh, "A parallel hybrid banded system solver: the SPIKE algorithm," *Parallel Computing*, vol. 32, no. 2, pp. 177–194, 2006.

[3] E. Polizzi and A. Sameh, "SPIKE: a parallel environment for solving banded linear systems," *Computers & Fluids*, vol. 36, no. 1, pp. 113–120, 2007.

[4] M. Manguoglu, F. Saied, A. Sameh, and A. Grama, "Performance models for the SPIKE banded linear system solver," in *Proceedings of the 2010 Ninth International Symposium on Parallel and Distributed Computing (ISPDC)*, IEEE, Istanbul, Turkey, July 2010.

[5] X. Wang, Y. Xu, and W. Xue, "A hierarchical tridiagonal system solver for heterogenous supercomputers," in *Proceedings of the 2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, IEEE, New Orleans, LA, USA, November 2014.

[6] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W.-M. W. Hwu, "A scalable, numerically stable, high-performance tridiagonal solver using GPUs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society Press, Salt Lake City, UT, USA, November 2012.

[7] H. Gabb, "Intel® adaptive SPIKE-based solver," Technical report, Intel, Santa Clara, CA, USA, 2010.

[8] L. W. Chang and W. M. Hwu, *A Guide for Implementing Tridiagonal Solvers on GPUs*, Springer, Berlin, Germany, 2014.

[9] J. Mair, Z. Huang, D. Eyers, and Y. Chen, "Quantifying the energy efficiency challenges of achieving exascale computing," in *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 943–950, IEEE, Shenzhen, China, May 2015.

[10] M. U. Ashraf, F. Alburaei Eassa, A. Ahmad Albeshri, and A. Algarni, "Performance and power efficient massive parallel computational model for HPC heterogeneous exascale systems," *IEEE Access*, vol. 6, pp. 23095–23107, 2018.

[11] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Solving tridiagonal linear systems using field programmable gate arrays," in *Proceedings of the 4th International Conference on*

Computational Methods (ICCM 2012), Gold Coast, QLD, Australia, November 2012.

[12] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Comparison of high level FPGA hardware design for solving tri-diagonal linear systems," *Procedia Computer Science*, vol. 29, pp. 95–101, 2014.

[13] S. Palmer, *Accelerating Implicit Finite Difference Schemes Using a Hardware Optimised Implementation of the Thomas Algorithm for FPGAs*, Cornell University, Ithaca, NY, USA, 2014.

[14] H. Macintosh, D. Warne, N. A. Kelson, J. Banks, and T. W. Farrell, "Implementation of parallel tridiagonal solvers for a heterogeneous computing environment," *The ANZIAM Journal*, vol. 56, pp. 446–462, 2016.

[15] B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, "Chapter 2—introduction to OpenCL," in *Heterogeneous Computing with OpenCL*, B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, Eds., pp. 15–38, Morgan Kaufmann, Burlington, MA, USA, 2nd edition, 2013.

[16] B. P. Flannery, S. Teukolsky, W. H. Press, and W. T. Vetterling, *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, Cambridge University Press, Cambridge, UK, 1992.

[17] P. Arbenz, A. Cleary, J. Dongarra, and M. Hegland, "A comparison of parallel solvers for diagonally dominant and general narrow-banded linear systems II," in *Euro-Par'99 Parallel Processing*, pp. 1078–1087, Springer, Berlin, Germany, 1999.

[18] C. R. Dun, M. Hegland, and M. R. Osborne, "Parallel stable solution methods for tridiagonal linear systems of equations," in *Proceedings of the Computational Techniques and Applications Conference (CTAC95)*, pp. 267–274, World Scientific Publishing, River Edge, NJ, USA, August 1996.

[19] M. Hegland, "On the parallel solution of tridiagonal systems by wrap-around partitioning and incomplete *LU* factorization," *Numerische Mathematik*, vol. 59, no. 1, pp. 453–472, 1991.

[20] C. C. K. Mikkelsen and M. Manguoglu, "Analysis of the truncated SPIKE algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 4, pp. 1500–1519, 2008.

[21] Intel Corporation, "Developer guide for Intel® Sdk for openCL™ applications," 2018, https://software.intel.com/en-us/openclsdk-devguide-2017.

[22] K. Mendiratta, "a banded SPIKE algorithm and solver for shared memory architectures," Masters' thesis, University of Massachusetts, Amherst, MA, USA, 2011.

[23] Intel Corporation, "?dtsvb," 2018, https://software.intel.com/en-us/mkl-developer-reference-c-dtsvb.

[24] J. D. McCalpin, "STREAM: sustainable memory bandwidth in high performance computers," Technical report, University of Virginia, Charlottesville, VA, USA, 1995.

[25] J. McCalpin, *Memory Bandwidth and Machine Balance in High Performance Computers*, IEEE Technical Committee on Computer Architecture Newsletter, 1995.