The Strong Scaling Advantage of FPGAs in HPC for N-body Simulations

JOHANNES MENZEL, CHRISTIAN PLESSL, and TOBIAS KENTER, Paderborn University,
Department of Computer Science and Paderborn Center for Parallel Computing, Germany

N-body methods are one of the essential algorithmic building blocks of high-performance and parallel computing. Previous research has shown promising performance for implementing n-body simulations with pairwise force calculations on FPGAs. However, to avoid challenges with accumulation and memory access patterns, the presented designs calculate each pair of forces twice, along with both force sums of the involved particles. Also, they require large problem instances with hundreds of thousands of particles to reach their respective peak performance, limiting the applicability for strong scaling scenarios. This work addresses both issues by presenting a novel FPGA design that uses each calculated force twice and overlaps data transfers and computations in a way that allows to reach peak performance even for small problem instances, outperforming previous single precision results even in double precision, and scaling linearly over multiple interconnected FPGAs. For a comparison across architectures, we provide an equally optimized CPU reference, which for large problems actually achieves higher peak performance per device, however, given the strong scaling advantages of the FPGA design, in parallel setups with few thousand particles per device, the FPGA platform achieves highest performance and power efficiency.

CCS Concepts: • **Hardware** → **Hardware accelerators**; • **Computing methodologies** → *Massively parallel and high-performance simulations*; • **Computer systems organization** → *Reconfigurable computing*;

Additional Key Words and Phrases: FPGA, OpenCL, high level synthesis, N-body simulations, strong scaling, arithmetic units

ACM Reference format:

Johannes Menzel, Christian Plessl, and Tobias Kenter. 2021. The Strong Scaling Advantage of FPGAs in HPC for N-body Simulations. *ACM Trans. Reconfigurable Technol. Syst.* 15, 1, Article 10 (November 2021), 30 pages. <https://doi.org/10.1145/3491235>

1 INTRODUCTION

During the past couple of years, **Field-Programmable Gate Arrays (FPGAs)** have started to receive increased attention as accelerators in compute and data centers due to their power efficiency

Johannes Menzel and Tobias Kenter contributed equally to this research.

This work is partially funded by the Federal Ministry of Education and Research (BMBF) and the state of North Rhine-Westphalia as part of the NHR Program. The authors gratefully acknowledge the funding of this project by computing time provided by the Paderborn Center for Parallel Computing (PC²).

Authors' address: J. Menzel, C. Plessl, and T. Kenter, Paderborn University, Department of Computer Science and Paderborn Center for Parallel Computing, Warburger Str. 100, Paderborn, North Rhine-Westphalia, Germany, 30098; emails: jmenzel@mail.uni-paderborn.de, {christian.plessl, kenter}@uni-paderborn.de.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2021 Copyright held by the owner/author(s).

1936-7406/2021/11-ART10 \$15.00

<https://doi.org/10.1145/3491235>

and dependable performance [6, 34]. Targeting scientific simulations, FPGA acceleration work includes stencil computations [21, 37, 45], computational fluid dynamics [19, 20] and shallow water simulations [22], dense [12, 30] and sparse [18] linear algebra, and FFTs [36, 39]. N-body methods, used in molecular dynamics and astronomy, are another an important class of parallel algorithms for **high-performance computing (HPC)** that has been considered for FPGA acceleration [8], recently even in a multi-FPGA setup [15].

The key operation for N-body methods is the calculation and accumulation of forces acting on each particle. Existing FPGA implementations [8, 15] perform this individually per target particle, such that for each pair of particles the force between them is calculated twice, with both involved particles taking the role of accumulation target once. In contrast, in optimized CPU implementations [32, Chapter 6] the calculated force would be reused for accumulation to both targets. Another shortcoming we saw for existing FPGA work is that they only pipeline the central computation phase and around that contain sequential steps to setup tile buffers or distribute target particles. This allows them to reach their respective peak performance only for large numbers of particles beyond 100K, where the streaming phase of computation dominates the overall execution time.

In this work, we present a new design and implementation that resolves these two challenges. The introduced symmetric accumulation scheme around 2×2 force calculation blocks allows our FPGA design to scale to the arithmetic resource limits of the device while matching the algorithmic efficiency possible on CPUs. Also, by fully overlapping all phases of a complete simulation, even small problem sizes of around 1,000 particles suffice to reach the peak performance on one FPGA. This opens up potential for multi-device scaling, where problems with tens or hundreds of thousands of particles can efficiently be distributed to multiple FPGAs. To this end, we implemented a ring solver architecture as suggested by Pacheco [32, Chapter 6], realizing the required communication structure with optical point-to-point links between FPGAs.

To perform a fair and detailed comparison to a CPU reference, we implemented the same algorithmic structure for CPU. Furthermore, diligent optimization for the CPU version is performed, using compiler intrinsic functions to achieve full SIMD vectorization and efficient usage of vector registers. Also, an architectural bottleneck around special function arithmetic for inverse square roots was resolved using an iterative refinement scheme. We see that operating close to its arithmetic peak performance, a dual socket compute node with 20-core Xeon Gold 6148 CPUs outperforms a dual FPGA Stratix 10 GX2800 design at similar power efficiency because the FPGA hardware trails in terms of double precision arithmetic potential. However despite the same communication structure, to operate near peak performance, the CPU design requires around 32,000 particles per node. This is particularly relevant when considering strong scaling to multiple nodes, where for example for an 18,432-element problem, a multi-FPGA setup consisting of 12 nodes is by more than $3\times$ the fastest platform, for a 6,144-element problem even by $10\times$.

To summarize, the specific contributions of this work include:

- A novel FPGA design that allows for the first time to efficiently use the symmetry of forces in n-body problems on FPGA while at the same time scaling the utilized arithmetic up to the resource limits of the device.
- A comprehensively optimized CPU implementation with full vectorization, throughput-optimized calculation of inverse square roots, and manual interleaving of loop iterations to mitigate register pressure.
- A multi-device ring solver for both target platforms.
- Analysis of performance for different problem sizes, comparisons to related work, and implications for future directions.
- Full open source availability of both designs at <https://github.com/pc2/n-body-ring-solver>.

The rest of the article is organized as follows: Section 2 discusses the background and scope of this work in more detail. In Section 3, we present our FPGA design under three aspects. First, we analyze the arithmetic requirements in relation to the device-specific resource limitations. Second, we present the novel force calculation and accumulation scheme. Third, we discuss the pipelining and communication infrastructure that allows to keep the expensive force calculation arithmetic well occupied even for problem sizes as small as 768 particles per device. Section 4 introduces our equally optimized CPU implementation with a focus on vectorization and high utilization of the arithmetic units available. Both implementations are then compared in terms of performance and scalability in Section 5. We summarize the results in Section 6 and compare them to previous state-of-the-art implementations on all architecture types (CPU, GPU, FPGA, ASIC) before concluding in Section 7.

2 BACKGROUND AND APPLICATION CONTEXT

The computational core of n-body methods is formed by the calculation of forces f_{ij} , denoting the force that particle j exerts on particle i . These particles can be celestial bodies or entire galaxies interacting via gravitational forces, or atomic particles interacting among others via electrostatic forces. For both cases, the force calculation takes the form

$$\vec{f}_{ij} = G \cdot \frac{m_i \cdot m_j}{|\vec{s}_j - \vec{s}_i|^3} \cdot (\vec{s}_j - \vec{s}_i), \quad (1)$$

with vectors $\vec{s}_i, \vec{s}_j$ denoting the positions of particles i, j in space, and in the gravitational case m_i, m_j denoting the masses of i, j , and G the gravitational constant. The corresponding force that particle i exerts on particle j is equal in magnitude and acts in the opposite direction, i.e.,

$$\vec{f}_{ji} = -\vec{f}_{ij}, \quad (2)$$

which can be seen from Equation (1) and corresponds to Newton's third law of motion. To calculate the total force exerted on particle i , the forces from all other particles need to be summed up.

$$\vec{F}_i = \sum_{j=0, j \neq i}^{N-1} \vec{f}_{ij} \quad (3)$$

In an integration step, also denoted as timestepping, $\vec{F}_i$ is used to update the velocity and position of the particle. The most straightforward integration is the forward Euler method, however, leap-frog or higher order symplectic integrators [44] are typically used in practice for increased accuracy and more stable long-term behavior.

For a system of n particles, the outlined force calculation of $\vec{F}_i$ based on all other particles $j \neq i$ has the time complexity $O(n^2)$. Exploiting the symmetry by using each calculated force $\vec{f}_{ij}$ for both force sums $\vec{F}_i, \vec{F}_j$ can save a constant factor of 2 in force calculations at identical accuracy, also denoted as reduced solver. Popular methods like the tree-based Barnes-Hut [3] can reduce the time complexity of force calculation to $O(n \cdot \log(n))$, at accuracy trade-offs that can, however, be controlled by the algorithm. With the same type of tradeoff, the fast multipole method [13] can even achieve $O(n)$ complexity and has been shown to pay off for large numbers of particles [26].

2.1 Application Context and Scope

N-body methods are one of the popular *Berkeley Dwarfs* [2] in the landscape of parallel computing and represent a fundamental pattern that occurs in multiple scientific domains, such as cosmological simulations, plasma physics, and **molecular dynamics (MD)**. In this work, we focus on the

general suitability of modern FPGAs as accelerators in a parallel HPC setup for offloading this type of method. We define the following scope for our study:

- We implement direct pair-wise force interactions with a reduced solver exploiting Newton's third law. As approximate algorithms with lower time complexity exist, this puts a special emphasis on problems with high accuracy requirements and setups that can still reach their target performance per timestep this way.
- The calculations use the gravitational type of force terms as in Equation (1). These are long-range forces that remain above numerical zero even at long distances.
- All positions, velocities, and forces are stored and calculated in double precision to match the high accuracy setup.
- A leapfrog-based symplectic integrator with variable higher order is also used to match the accuracy considerations. Configurable order at runtime allows for flexible timestepping without repeated FPGA synthesis.

Thus, while the focus of our evaluation is on architectural suitability for the general method, the chosen setup is closer to cosmological simulations than to MD. In MD, the long-range electrostatic forces take the same form, but are more efficiently calculated with the FFT-based Particle Mesh Ewald [7, 9] method that comes along with the periodic boundary conditions that are desired for MD simulations. The force term itself could also easily be adapted to the Lennard-Jones potential, however, as this force expression decays with the powers of 6 and 12 over the radius, it actually becomes 0 beyond a certain cut-off radius and thus is better implemented along with some form of neighbor-lists to avoid excessive needless calculations. Comparing this work, however, to full cosmological simulations codes such as GADGET-2 [38], NBODY6++GPU [41], and ϕ -GPU [4], beyond additional physics specific to the domain, a typical addition for this use case would be adaptive timesteps along with a separation of forces into *regular* [41] all-to-all forces and more frequently calculated *irregular* [41] forces based on neighbor lists.

NBODY6++GPU and its predecessors as well as ϕ -GPU use complete all-to-all pairs for the regular forces, which matches our work directly. Their GPU-based all-to-all force calculations struggle to scale over multiple devices [4, 5, 41] for less than several tens of thousands of particles per GPU, compared to which the scalability of our FPGA design demonstrates an improvement. However, GADGET-2, as well as the PEPC code [42] that is additionally used for plasma physics both employ Barnes-Hut-type tree algorithms. For those, our implementation could serve as a building block for forces inside tree nodes as proposed in Reference [31]. Such a setup is also discussed by Makino and Daisaka [29] when introducing the GRAPE-8 ASIC for gravitational force calculation. They report to reach around half of their respective peak performance for around 20,000 particles per board with two ASICs and were aiming for future communication improvements that allow to lower this requirement to around 1,000 particles per board for usage within a tree algorithm, a goal that our FPGA design reaches.

3 FPGA DESIGN AND IMPLEMENTATION

From a high-level perspective, the presented FPGA design contains two major areas of contribution: first, an efficient force computation architecture that can scale to the resource limits of a single FPGA, and second, a timestepping and communication pipeline to scale to multiple devices. After a brief introduction to the target FPGA, we present these two design aspects in Sections 3.2 and 3.4, respectively.

3.1 Target FPGAs and OpenCL Design Entry

Many of the design choices and implementation patterns presented in this section are correlated with the target platform and programming environment that we introduce here, but the concepts

Fig. 1. Data flow graph (DFG) of a force calculation unit (FCU) with 5 sub/add units, 10 mul units, and 1 *rsqrt* unit.

are transferable to other FPGAs. The FPGAs used for this work are *Intel Stratix 10 GX2800*. They are integrated on *BittWare 520N* PCIe accelerator boards. Each of these boards provides four QSFP links that are connected in a ring topology with neighboring cards, providing two links to a previous and two links to a next FPGA board.

The FPGA designs and invoking host applications are entirely implemented in OpenCL, using the Intel FPGA SDK for OpenCL in version 20.4.0 to translate the OpenCL kernel code in a **high-level synthesis (HLS)** step into the specification of a spatial hardware pipeline in **hardware description language (HDL)**. After reviewing design reports and testing in emulation, this specification is translated into a bitstream for the actual FPGA with Quartus 19.4.0 as the backend synthesis tool. This matches the version of the board support package provided by Bittware that includes interfaces to PCIe and DDR memory, as well as to the QSFP links, which are exposed as serial point-to-point connections denoted as channels in the OpenCL programming environment.

The Stratix 10 GX2800 FPGAs contain **digital signal processing blocks (DSPs)** that can perform entire arithmetic operations like multiplication, addition, subtraction, and multiply-add on fixed-point or single precision (32 bit) floating point operands [17]. This would make a single precision implementation the best fit for the target hardware. However, given the high accuracy requirements of the application, we implement the entire design in double precision to see if FPGAs can even compete in this domain.

3.2 Force Calculation Arithmetic

The fundamental building block for an n-body solver on current FPGAs¹ is formed by a force calculation datapath. Figure 1 illustrates the **force calculation unit (FCU)** realized in this work, using 3 subtractions for the position difference, 3 multiplications, 2 additions and 1 inverse square root for the distance calculation, 4 multiplications for the force value per unit vector, and 3 multiplications for the force vector. To calculate $\vec{F}_i$ (Equation (3)), each component of the 3-element output

¹Early FPGAs may not have had enough resources for the full arithmetic for one force calculation and would have used time-multiplexing of fewer arithmetic units.

Table 1. Characterization of Estimated Resource Consumption for Arithmetic in FCU and Reduction (RED)

Block	Component	ALUTs	FFs	DSPs	$\frac{DSPs}{ALUTs}$
FCU	3× sub	3,734	647	0	0
	3× mul + 2× add	4,367	5,881	9	$\frac{1}{485}$
	1× <i>rsqrt</i>	1,778	2,740	8	$\frac{1}{222}$
	7× mul	3,644	2,175	28	$\frac{1}{130}$
RED	3× add	4,419	5,091	0	0
Sum		17,942	16,534	45	$\frac{1}{399}$
Available		1,397K	2,794K	4,713	$\frac{1}{296}$

vector of a FCU is summed up in an accumulator. If multiple FCUs produce partial forces for $\vec{F}_i$, then a sum reduction over these FCUs' results can be performed before accumulation. It is important to note that the FCU datapath as well as any sum reduction can have a high latency without major drawback as long as they are fully pipelined with a throughput of one force-pair and one partial result per cycle. The accumulation, however, can be critical when new inputs arrive in each subsequent cycle.

Table 1 summarizes the estimated resource utilization of the described FCU and the additions for a **reduction (RED)** with the values of one neighboring FCU. The values are derived from the report generated after the HLS step of the Intel FPGA SDK for OpenCL. The default target frequency of 480 MHz is used and all units are fully pipelined with an **initiation interval (II)** of 1 without setting explicit or implicit latency constraints. Since the later full hardware design builds upon the intermediate results of the HLS step, the resource estimation values give a good indication. Through backend optimizations, the resources differ a bit in the final design mapped to hardware, however, they are no longer reported on the granularity of single arithmetic units or even the full FCU, but only on complete kernel level. The resources reported here are **lookup-tables (ALUTs)**, registers (**FFs for flip-flops**), and DSPs. To implement the required double precision (64 bit) floating point arithmetic, DSPs have to be combined with or replaced by logic resources. The values in Table 1 reflect the choices made automatically by the HLS compiler. Several important observations can be made. First, the mix of resources utilized differs between different arithmetic operations. Addition/subtraction is implemented purely in logic, whereas multiplications and *rsqrt* use a mix of DSPs and logic. The combination of multiplication and addition that is marked by the compiler as size 3 dot product uses a different mix of resources than the pure subtractions, multiplications, and additions before and after it. Second, the *rsqrt* operation is the most resource-intensive single operation, but with 10% ALUT and 18% DSP usage small enough to not dominate the overall design. Third, the FF usage looks somewhat inconsistent, in particular when comparing the 3 subtractions and 3 additions in FCU and RED, respectively. This is related to pipeline latencies that the HLS compiler includes for the identified schedule of operations and can change depending on how the reduction is combined with the FCU. Thus, FF usage estimates need to be taken with a grain of salt at this stage.

Using the sum of resources projected for one FCU and one RED in relation to the total amount of resources that are available in the kernel region on the target FPGA, we see that a design with multiple FCUs can be expected to be limited by logic resources, as indicated by the ratios between DSPs and ALUTs. Based on the number of ALUTs, up to 77 FCU, RED pairs could be possible, compared to 104 blocks when only regarding DSPs. Utilization of FFs may allow for 168 blocks,

Table 2. Illustration of Triangular Force Pair Calculations

j	i									
	0	1	2	3	4	5	6	7	8	9
0		x	x	x	x	x	x	x	x	x
1	-		x	x	x	x	x	x	x	x
2	-	-		x	x	x	x	x	x	x
3	-	-	-		x	x	x	x	x	x
4	-	-	-	-		x	x	x	x	x
5	-	-	-	-	-		x	x	x	x
6	-	-	-	-	-	-		x	x	x
7	-	-	-	-	-	-	-		x	x
8	-	-	-	-	-	-	-	-		x
9	-	-	-	-	-	-	-	-	-	

however, with more uncertainty. In practice, the logic limitation of this type of double precision designs gets aggravated by two factors. On the one hand, the entire control flow on the FPGA is also implemented with logic resources; on the other hand, each 2 ALUTs and 4 FFs are combined into so-called **adaptive logic modules (ALMs)**. For operations with a good balance of ALUT and FF requirements, the ALMs can often be densely packed, however, less balanced building blocks or the spatial mapping of blocks on the device tend to leave some ALMs not fully occupied, such that the total ALM usage is higher than predicted by the maximum of ALUT and FF usage.

3.3 Force Calculation Loops and Accumulation

3.3.1 Triangular Loop Pattern. When exploiting the symmetry property as outlined in Equation (2), each force-pair $\vec{f}_{ij}, \vec{f}_{ji}$ needs to be calculated only once and can be accumulated both into $\vec{F}_i$ and $\vec{F}_j$, only with a negative sign or subtraction in the second case. In this section, we first describe the challenges of an implementation with a triangular loop structure and computation pattern before presenting our improved approach. We illustrate these challenges here specifically with regard to the HLS strategies of the Intel FPGA for OpenCL compiler, however, only the second described problem is somewhat specific to the HLS compilation, whereas accumulation latency and challenges with data access patterns are common to other FPGA design strategies. Listing 1 shows the simple triangular loop pattern in OpenCL kernel code, and Table 2 gives a visual representation. For the listing, we have encoded the three-dimensional position information $\vec{s}$ and the scalar mass m into a common struct p . In the table, x entries denote directly calculated and accumulated forces and $-$ entries denote forces taken from the corresponding transpose position and subtracted.

```

1 for(int i=0; i<n; i++) {
2     for(int j = i+1; j<n; j++) {
3         double3 f_ij = FCU(p[i], p[j]);
4         F[i] += f_ij;
5         F[j] -= f_ij;
6     }
7 }

```

Listing 1. Challenging triangular force calculation pattern.

Implementing this pattern on multi-threaded SIMD architectures is only a minor challenge of cyclic- or block-wise cyclic scheduling of the outer loop to the threads and loop-vectorization of the inner loop. Here, both the unvectorized loop peel and remainder, as well as the difference

in executed loop iterations between any two threads is only linear in size, and thus negligible compared to the quadratic number of loop iterations.

However, when implementing this pattern with the Intel FPGA SDK for OpenCL and other HLS Languages targeting loop-pipelined execution on an FPGA, three somewhat interrelated challenges come up. First, the distance between two subsequent additions into $\vec{F}_i$ and $\vec{F}_j$ is variable and becomes very small towards the last iterations of the nested loops, i.e., the lower right corner in Table 2. Since the latency of double precision addition on FPGA requires multiple cycles, this creates a dependency limiting the pipeline throughput of a thus implemented loop. The HLS compiler creates an initiation interval $II > 1$ (the time between scheduling two subsequent iterations of the loop) for which it can guarantee that the previous value $\vec{F}_i$ is available before the next $\vec{f}_{ij}$ is added. Since this type of design is statically scheduled, the same II is applied for all iterations of both loops, even though most of them have a much larger reuse distance of $\vec{F}_i$ and $\vec{F}_j$ values. The easiest design pattern [16, Chapter 6] does resolve this dependency and reduce the II to 1 by using a shift register and accumulating multiple partial sums in an interleaved mode and performing a reduction over those afterwards. However, this pattern can involve considerable resource overheads, in particular when applied to multiple sums in parallel. Our solution to this is presented in Section 3.3.3 and involves no such overheads.

Second, due to the different trip counts of the inner loop (over j , line 2 in Listing 1) for different iterations of the outer loop, only the inner loop can be automatically pipelined. This means that for each iteration i of the outer loop, the execution time t_{cj} in clock cycles of the inner loop can be modeled as $t_{cj} = n - i - 1 + \lambda_{FCU}$, with λ_{FCU} denoting the latency of the operations inside the loop. That means λ_{FCU} shows up for each iteration of the outer loop, summing up to $t_{ci} = \frac{n^2-n}{2} + n \cdot \lambda_{FCU}$ for the execution time t_{ci} of the outer loop. The high latency of the involved double precision arithmetic, in particular that of the *rsqrt* operation, causes high λ_{FCU} values in the order of hundreds of cycles. This represents a considerable overhead for small n or when additional blocking is applied to only operate on a subset of particles at a time. Therefore, we instead aim for a rectangular loop structure with fixed inner loop trip counts, a structure that allows to pipeline also the outer loop, such that the overhead λ_{FCU} only occurs once for the entire loop nest. We present this scheme next in Section 3.3.2.

A third challenge of the triangular loop pattern is the access pattern to particles p . When these particles are read from a local memory buffer, there is no problem to supply $p[i]$ and $p[j]$ to a simple pipeline with one single FCU. However, when utilizing more resources by unrolling one of the loops to instantiate more FCUs, different FCUs can not easily be supplied by different banks of the local memory buffer as the access pattern shifts by one per row (cf. Table 2). There are techniques to resolve this challenge individually by diagonal addressing schemes. We use such a scheme in Section 3.3.3 to address the accumulation latencies identified as a first challenge here, but the resolution to banked input patterns actually comes from the transition to a quadratic loop structure.

3.3.2 Quadratic Loop Pattern around 2×2 PEs. When transitioning from the triangular loop pattern to a quadratic one, we do not want to increase the total number of iterations that was improved by a factor of 2 due to the force symmetry. Instead, we increment each of two nested loops by groups of two particles i and j per step, thus processing four force-pairs per cycle, two via calculations in two FCU instances and two as symmetric transposed forces. Table 3 illustrates the chosen computation scheme. For each 2×2 group, when $i < j$, the pairs formed by both particles $p[j]$ with the first $p[i]$ are calculated directly, otherwise the pairs of both particles $p[i]$ with the second $p[j]$ are formed. To keep the accumulation scheme simple, forces are always calculated as

Table 3. Illustration of Improved Quadratic Force Pair Calculations in 2×2 Groups

j	i									
	0	1	2	3	4	5	6	7	8	9
0	0	+	+	+	+	+	+	+	+	+
1	-x	0	x	x	x	x	x	x	x	x
2	-x	-	0	+	+	+	+	+	+	+
3	-x	-	-x	0	x	x	x	x	x	x
4	-x	-	-x	-	0	+	+	+	+	+
5	-x	-	-x	-	-x	0	x	x	x	x
6	-x	-	-x	-	-x	-	0	+	+	+
7	-x	-	-x	-	-x	-	-x	0	x	x
8	-x	-	-x	-	-x	-	-x	-	0	+
9	-x	-	-x	-	-x	-	-x	-	-x	0

Fig. 2. 2×2 processing element (PE).

$\vec{f}_{ij}$ and subtracted in the lower left triangle. Transposing the illustration along the diagonal (here explicitly filled with 0s as a valid result is required for every position) shows that either in the upper right or lower left triangle, every position is once filled with an x marking explicit force calculation, and once with an accumulation sign (now $-$ or $+$) for using the transposed value.

Figure 2 shows the **processing element (PE)** that processes each 2×2 group of particles in a fully pipelined way. It generates outputs in two directions, i and j , such that accumulations are actually performed both horizontally and vertically like in the triangular scheme (Listing 1, lines 4 and 5). In addition to the two FCUs, the PE also contains one reduction block that is needed to add up the two calculated entries that belong to the same particle, e.g., Table 3 column $i = 0$ or row $j = 1$. For the respective other particle (column $i = 1$ or row $j = 0$), zero forces are returned in this direction with combined values provided via the transposed block in the other direction. Additionally, Figure 2 shows the six input or output selection blocks that are actually implemented as ternary expressions in OpenCL and realized as multiplexers on FPGA. Compared to the resource intensive arithmetic, these simple two-input multiplexers are small.

Table 4. Illustration of Accumulation Sequence

j	i									
	0	1	2	3	4	5	6	7	8	9
0										
1	0		5		10		15		20	
2										
3	21		1		6		11		16	
4										
5	17		22		2		7		12	
6										
7	13		18		23		3		8	
8										
9	9		14		19		24		4	

With the thus simplified quadratic loop structure around the PEs, the two nested loops can easily be fused into a single pipelined loop with quadratic trip count, thus incurring the overhead of pipeline latency only once for all iterations. Also, when creating multiple 2×2 PE instances via loop unrolling, the required data per PE belongs to disjoint pairs of columns or rows in Table 3 and thus can automatically be put in multiple banks of local memory. However, before proceeding to the complete force calculation pipeline, we first discuss the accumulation scheme at the hand of a single PE.

3.3.3 Accumulation Scheme. As discussed in Section 3.3.1, hiding the latency of accumulations with shift registers can be expensive in terms of resources. When we can schedule the accumulation to different force sums $\vec{F}_i$ and $\vec{F}_j$ in an interleaved way, such that each individual sum is used again after a guaranteed latency of λ_{acc} cycles, a pipeline with $II = 1$ is possible without this overhead. Table 4 illustrates how this is achieved with a diagonal computation pattern that is realized through corresponding index calculations inside the manually fused loop nest. When calculating $b \times b$ blocks of 2×2 with one PE, a reuse distance of $b - 1$ iterations can be guaranteed to the HLS compiler with an annotation. In Table 4 with $b = 5$, the reuse distance in columns is 4 steps, e.g., for columns $i = 2, 3$ between step 1 and 5. In practice, we use $b = \lambda_{acc} = 16$ for a requirement of 32 particles per block (a smaller latency of around 14 would work, but requires more logic for index calculations). Additionally instantiating multiple PEs in that loop leads to a corresponding factor in additional block size requirements. However, when a common factor is applied in both dimensions, the required number of particle needs to be multiplied only once with this factor. Concretely, a total of $16 \cdot 2 \times 2$ PEs in a 4×4 layout (for a total of 32 FCUs) requires only a block size of $b = 2 \cdot 4 \cdot \lambda_{acc} = 128$ for the required guarantee on reuse distance.

3.4 Kernels and Communication Pipeline

3.4.1 Kernel Structure. Figure 3 illustrates the kernel structure of the proposed design, with an array of PEs and the accumulation scheme around them instantiated in the central *Force Calculation* kernel and further infrastructure around this kernel. The key objective for the complete design is to keep these PEs in the *Force Calculation* kernel occupied at all times. This is achieved by, on the one hand, overlapping the computation with communication within a timestep and, on the other hand, overlapping consecutive timesteps. For the former, in total four kernels are connected to

Fig. 3. Full computation pipeline including communication.

the *Force Calculation* kernel, two at the input side, namely, the *Remote-* and *Local Particle Buffers*, and two at the output, namely, the *Remote-* and *Local Force Caches*. For the latter, the *Integrator* kernel handles receiving forces from the *Force Caches* and writing thereby updated particles to the particle buffers simultaneously.

3.4.2 Communication Interface and Dataflow. Communication between kernels as well as communication to remote FPGAs, both illustrated by the arrows in Figure 3, is implemented with the Intel channel extension, a variant of OpenCL pipes. These channels are realized as FIFO buffers, in our design using BRAM resources such that each channel can contain at least one full block of particles before stalling. Channels within the same FPGA only add a single cycle of latency, whereas the physical data transfer and protocol used for communication with neighboring FPGAs in the ring topology lead to around 150 cycles latency, depending on the clock frequency. In Figure 4, we illustrate how blocks are streamed through the kernel structure for a minimal example with only four blocks distributed to two FPGAs. In the following sections, we discuss the roles of individual kernels while following the flow of blocks through the pipeline.

3.4.3 Particle Buffer Kernels. In the distributed calculation following Pacheco [32, Chapter 6], each remote block is combined with each local block owned by the respective device. The remote particle buffer forwards one block of particles to its next neighbor in the communication ring and then receives one block from the previous neighbor. To forward a completed remote block as early as possible to the next device, each remote block is matched with all local blocks before work on the next remote block begins. This can be observed in Figure 4(a–d): In cycle 0, the *Remote Particle Buffer* sends the first remote block to the *Force Calculation* kernel and immediately afterwards forwards a copy of it to the next device. On the *Local Particle Buffer* side, starting from cycle 128, for each remote block all local blocks are sent to the *Force Calculation* kernel for computation. For the remote blocks received from the second device, the *Remote Particle Buffer* stalls until the compute kernel is free again to accept new remote blocks. In terms of complexity, in a simulation with K devices and each device holding B blocks, the number of active clock cycles in the *Remote Particle Buffer* is in $O(K \cdot B)$, while the *Local Particle Buffer* is in $O(K \cdot B^2)$.

3.4.4 Force Calculation Kernel. Fed by the particle buffer kernels is the *Force Calculation* kernel, which contains the 2×2 force calculation PEs and accumulation scheme introduced in Section 3.3.

Fig. 4. Model of block-wise dataflow through kernel structure with two devices and two blocks per device. Only the dataflow of the first device is illustrated. The blue and orange blocks are local to the first device, while the green and red blocks are local to the second device. A stripe pattern from top-left to bottom-right indicates a local block, the opposite stripe pattern orientation indicates a remote block. A crossed stripe pattern indicates that local and remote blocks are handled simultaneously. For readability, the latencies are not representative of the real kernel structure, but fixed to 64 cycles for every type of operation.

Listing 2 illustrates the loop pattern to implement this kernel and connect it to the surrounding design. The loops of lines 29 and 31 create with the help of unrolling attributes a configurable 2D array of the 2×2 PEs that are introduced in Section 3.3.2 and Figure 2.

The nested loops of lines 9 and 16 implement the pattern of slow iterations over remote blocks and fast iterations over local blocks, with concrete buffers being filled in lines 19ff for local and lines 12ff for remote blocks using `read_channel_intel` and `write_channel_intel` statements for reading, respectively, sending one element (particle struct, respectively, force vector) per cycle. To keep the PEs occupied, this loading needs to overlap with the computation. The specific pattern of first declaring, then filling, and then using local buffers like `l_particles` inside one surrounding loop enables the Intel FPGA SDK for OpenCL, by creating multiple private copies of the buffer, to also pipeline that outer loop, that is, to achieve dataflow between the individually pipelined sub-loops (e.g., lines 19, 26, 40, 44). For example, inside the loop in line 16, while a new instance of `l_particles` is filled in lines 19ff, concurrently the previous instance is used for force calculation in the block starting in line 26. Overall, this leads to a transparent form of task-level parallelism for overlapping communication with computation, in this listing with four concurrent streaming tasks (12ff, 19ff, 40ff, and 44ff) and the computation task (line 26ff). The resulting communication and computation can again be observed in Figure 4(a,d–g): Before the first computation block starts in cycle 256, the *Force Calculation* kernel already receives both the first local- and remote block from the *Particle Buffers*. Then, during the force calculation for those, the second local block is streamed in, such that the calculation of the next pair can begin once the force pipeline is free. During the combination of the second local block with the first remote block, the partial local- and remote forces resulting from calculation of the first blocks are streamed out to the *Force Caches*, while again the first local- and second remote block is streamed in for the next calculation. This pattern continues until all remote blocks in the system are combined with all blocks local to the device.

Looking further at the at the loop around the array of PEs, we note that the loop in line 26ff is running for λ_{acc}^2 iterations (denoted as `LAT_ACC` in the listing). The diagonal indexing as described in Section 3.3.3 is derived from variable `bb` and leads to a reuse distance of $\lambda_{acc} - 1$ that is provided to the compiler in line 25. Along with the illustration (Figure 4(d,e)), this reveals an important

```

1 // configuration of PE array
2 #define L_PEs 4
3 #define R_PEs 4
4 #define LAT_ACC 16
5 #define L_BLK_SIZE (2 * L_PEs * LAT_ACC)
6 #define R_BLK_SIZE (2 * R_PEs * LAT_ACC)
7 // placeholder: skipped outer loop over time steps here
8 // process all remote blocks
9 for(uint r_blk=0; r_blk<r_blocks; r_blk++) {
10     particle_t r_particles[R_BLK_SIZE];
11     // load remote block
12     for(uint rr=0; rr<R_BLK_SIZE; ++rr) {
13         r_particles[rr] = read_channel_intel(chan_r_particle_buff_to_comp);
14     }
15     // process all local blocks
16     for(uint l_blk=0; l_blk<l_blocks; l_blk++) {
17         particle_t l_particles[L_BLK_SIZE];
18         // load local block
19         for(uint ll=0; ll<L_BLK_SIZE; ++ll) {
20             l_particles[ll] = read_channel_intel(chan_l_particle_buff_to_comp);
21         }
22         double3 r_forces[R_BLK_SIZE];
23         double3 l_forces[L_BLK_SIZE];
24         // sequential computation steps per block
25         #pragma ivdep safelen(LATENCY_FACTOR-1)
26         for(uint bb=0; bb<LAT_ACC*LAT_ACC; ++bb) {
27             // parallel PE array
28             #pragma unroll
29             for(uint lp=0; lp<L_PEs; lp++){
30                 #pragma unroll
31                 for(uint rp=0; rp<R_PEs; rp++){
32                     // placeholder: individual 2x2 PE here
33                 }
34             }
35             // placeholder: reduction and accumulation in local direction here
36             // placeholder: reduction and accumulation in remote direction here
37             // reuse distance LAT_ACC-1 for both accumulations
38         }
39         // write local forces
40         for(uint ll=0; ll<L_BLK_SIZE; ++ll) {
41             write_channel_intel(chan_compute_to_l_force_cache, l_forces[ll]);
42         }
43         // write remote forces
44         for(uint rr=0; rr<R_BLK_SIZE; ++rr) {
45             write_channel_intel(chan_compute_to_r_force_cache, r_forces[rr]);
46         }
47     }
48 }

```

Listing 2. Loop pattern for force calculation kernel. Innermost PE loops are unrolled, all others pipelined. Using automatically generated private copies of buffers, loop nests in lines 12, 19, 26, 40 and 44 can overlap in time.

relationship between the block size b and the accumulation latency λ_{acc} . Obviously, $b \leq \lambda_{acc}^2$ should be ensured, as otherwise the data streams would stall the PE array. However, the accumulation latency requires $b = 2 \cdot \max(l_{dim}, r_{dim}) \cdot \lambda_{acc}$, which yields the following constraint on the latency: $\lambda_{acc} \geq 2 \cdot \max(l_{dim}, r_{dim})$. When these two constraints clash, one way to relax the first constraint, would be to stream multiple elements (particles respective forces) per cycle. However, for the designs realized in this work, the latency constraints given by the double precision implementation of the Intel FPGA SDK for OpenCL are high enough to satisfy the original constraint.

3.4.5 Force Cache Kernels. After the combination of a local- and a remote block finishes in the *Force Calculation*, the resulting partial forces have to be accumulated into the corresponding parts in the entire local- and remote force array. Since the local blocks are swapped out for every

combination on the input side, accumulation into a different block of the entire force array takes place in the *Local Force Cache* for every consecutive block. Oppositely, in the *Remote Force Cache*, first all interactions on the same remote block are accumulated before accumulation into the next block starts. Here, as soon as the accumulation into a remote block finishes, it can be sent to the next device, while simultaneously a new remote block is received from the previous device. Both patterns can be observed in Figure 4(f–j). The distinction into a cache and communication kernel is made to relax the memory dependency caused by the accumulation into a block and the replacement of that block by the one received from the next device. In any timestep, the accumulation into a local block is complete as soon as the last remote block was combined with it and thus the result can be streamed to the integrator immediately. In contrast, the accumulation into a remote block is complete after it was received from the previous device, since by then it was combined with all local blocks on all devices. For higher amounts of local blocks, this occurs much earlier than the completion of the local blocks. Consequently, the results of the *Remote Force Cache* are not sent to the integrator immediately, but are held back until the *Force Calculation* kernel begins the combination with the last remote block, such that both *Force Caches* stream the complete force array to the Integrator simultaneously (see Figure 4(f,i,k,l)).

3.4.6 Integrator Kernel. The integrator itself implements a basic step that updates both the velocities and the positions with the leapfrog method. To demonstrate that the design can be adapted to higher-order schemes, we have added support for sub-steps with individual coefficients for the updates as proposed by Kinoshita et al. [24] for up to 6th order and coefficients for 8th order published by Yoshida [44]. Configured by the host with the number of sub-steps and coefficients, it performs $(2k)$ th order symplectic integration with 2^k sub-steps, e.g., 6th order with 8 sub-steps, similarly discussed by Hut and Makino [14]. Due to memory dependencies within this scheme, the integrator is currently the only kernel that is not fully pipelined over multiple timesteps, but only over the blocks within each timestep. In practice, this type of dependency for the compiler does not impact the performance, not even for small problem sizes, because the latency of a single block traveling through the kernel structure is greater than the latency of finishing a timestep in the *Integrator*.

Currently, the *Integrator* kernel is designed to overlap timesteps in the sense that while particles and forces of the current timestep are still processed within the kernel structure, the *Integrator* can already partially calculate the set of particles for the next timestep. This overlap occurs, since the first local- and remote force blocks finish all while later local blocks are still combined with the last remote block. The action on these first blocks can thereby be immediately calculated and streamed to the particle buffers for the next timestep. This is visible in Figure 4(a,d,m), where in cycle 2,176 the integrator receives the first block of local- and remote forces and with it computes the corresponding block of particles for the next timestep, all while the combination of the last local- and remote block is still in action in the *Force Calculation* kernel. The gap between the *Force Calculation* kernel between timesteps is thus made smaller, as the combination of the first local- and remote block of the next timestep can already start during the time integration of the second block. The gap ultimately vanishes for higher amounts of blocks.

4 CPU IMPLEMENTATION

Our CPU implementation is a vectorized variant of Pacheco’s textbook reference [32, Chapter 6] and as such it is implementing the reduced triangular force accumulation scheme. Alongside vectorization, our second addition is the blockwise execution to improve cache utilization for larger problem sizes and to ensure scaling within a single node.

4.1 Vectorization

The vectorization was carried out by hand using Intel intrinsics and was specifically done with the Skylake server architecture of the Intel Xeon Gold 6148 in mind. This architecture contains two scheduler ports for arithmetic (Advanced Vector Extensions 512) AVX512 instructions, one for all types of arithmetic operations and one additional port for **(Fused-Multiply-Accumulate)** FMA-like operations. Similarly, the two ports are also available for the older AVX2 instructions.

4.1.1 Vectorization Pattern and Instruction Mix. In the most basic scheme, all force-pairs f_{ij} are calculated for each local particle p_i with all remote particles p_j if $i < j$. For vectorization, a single local particle p_i is spread over three AVX registers, one for each directional component, such that it may be combined with all necessary particles p_j . The exerting force on particle p_i is also spread over three AVX registers and only reduced to a single force after completing all force interaction calculations with this particle. Due to this triangular arrangement of the force-pairs and the unconstrained upper bound of the problem size n , the vectorization is split into peel, body, and remainder, where $O(n \cdot AVX_{width})$ unvectorized operations are performed for the peel and remainder, respectively. Still, the $O(n^2)$ vectorized operations in the loop body outweigh the impact of the peel and remainder. The vectorized loop body represented in Listing 3 contains 1 *sqr*t instruction, 1 *div* instruction, and 15 FMA-like instructions.

```

1  .L1:
2      //diff = pos[i] - pos[j]
3      vsubpd  zmm6, zmm20, ZMMWORD PTR [rdx+rbx*8]
4      vsubpd  zmm7, zmm21, ZMMWORD PTR [rax+rbx*8]
5      vsubpd  zmm5, zmm18, ZMMWORD PTR [rcx+rbx*8]
6      //dist^2 = dot_prod(diff,diff)
7      vmulpd  zmm0, zmm6, zmm6
8      vmovapd zmm14, zmm7
9      vfmadd231pd  zmm0, zmm7, zmm7
10     vfmadd231pd  zmm0, zmm5, zmm5
11     //dist = sqrt(dist^2)
12     vsqrtpd zmm0, zmm0
13     //dist^3 = dist * dist * dist
14     vmulpd  zmm10, zmm0, zmm0
15     vmulpd  zmm0, zmm10, zmm0
16     vmovapd zmm10, zmm6
17     //force_ij = (-G * mass[k] * mass[q])/dist^3
18     vmulpd  zmm4, zmm16, ZMMWORD PTR [r11+rbx*8]
19     vdivpd  zmm0, zmm4, zmm0
20     vmovapd zmm4, zmm5
21     //force[j] -= force_ij * diff
22     vfnmadd213pd  zmm14, zmm0, ZMMWORD PTR [r12+rbx*8]
23     vfnmadd213pd  zmm10, zmm0, ZMMWORD PTR [r14+rbx*8]
24     vfnmadd213pd  zmm4, zmm0, ZMMWORD PTR [r15+rbx*8]
25     //force[i] += force_ij * diff
26     vfmadd231pd  zmm13, zmm7, zmm0
27     vfmadd231pd  zmm12, zmm6, zmm0
28     vfmadd231pd  zmm11, zmm5, zmm0
29     //store back force[j]
30     vmovapd ZMMWORD PTR [r12+rbx*8], zmm14
31     vmovapd ZMMWORD PTR [r14+rbx*8], zmm10
32     vmovapd ZMMWORD PTR [r15+rbx*8], zmm4
33     add     rbx, 8
34     cmp     rbx, rdi
35     jb     .L1

```

Listing 3. Assembler listing of initial vectorized force calculation.

The Intel Xeon Gold 6148 CPU is well optimized for the FMA-like instructions with a throughput of one vector instruction per cycle for each of the two execution ports, regardless of the AVX

Table 5. Throughput (in CPI) and Latencies of Relevant AVX Instructions as Documented by Intel [1]

AVX-Type	Instruction	Latency	Throughput
AVX2	FMA-like	4	0.5
	<i>sqr</i> t	18	12
	<i>div</i>	14	8
AVX512	FMA-like	4	0.5
	<i>sqr</i> t	31	24
	<i>div</i>	23	16

extension used. However, the *sqr*t and *div* instructions are much less optimized. The latencies and throughputs (an inverse throughput metric corresponding to **cycles per instruction (CPI)**) of the relevant types of instructions are summarized in Table 5.

The maximum throughput of calculating the inverse square root can be achieved by hiding the latencies by chaining two independent *sqr*t instructions followed by two independent *div* instructions. This results in a throughput of 40 Cycles per inverse square root and 20 Cycles per inverse square root for the AVX512- and AVX2 instructions, respectively. In both cases this is enough to completely hide the other 15 FMA-like instructions behind the inverse square root calculation.

The performance of an implementation is measured in effective *MPairs/s*, that is, the number of force-pairs required by the direct solver calculated per second. The reduced solver uses each pair twice and has thereby a theoretical $2\times$ speedup per iteration against a direct solver. In a vectorized implementation, each iteration AVX_{width} force-pairs are calculated, but the clock frequency f_{max} is reduced when using larger AVX_{width} . Thus, the estimated peak performance of the vectorized implementations can be estimated as:

$$\begin{aligned}
 P_{AVX} &= 2 \cdot \frac{AVX_{width} \cdot f_{max}}{T_{inv\sqrt{}}} \\
 P_{AVX2} &= 2 \cdot \frac{4 \cdot 2,600\text{MHz}}{20} = 1,040\text{MPairs/s} \\
 P_{AVX512} &= 2 \cdot \frac{8 \cdot 2,200\text{MHz}}{40} = 880\text{MPairs/s}.
 \end{aligned}$$

Here, the higher AVX_{width} of the AVX512 implementation has no beneficial impact as the throughput of the *sqr*t and *div* instruction also doubles, while the maximum clock frequency is more limited. Consequently, the AVX2 implementation is faster by the ratio of the limiting clock frequencies.

4.1.2 Mitigation of the *Rsqrt* Bottleneck. Besides the presented normal approach of calculating the inverse square root, the Intel Xeon Gold 6148 also supports the lower precision *rsqr*t14 instruction. This instruction on its own, with its 14 bits of accuracy, would negate the benefit of using double precision accuracy, but the result can be refined using Newton's method, i.e.,

$$\begin{aligned}
 y_0 &= rsqr\text{t}14(x), \\
 y_{n+1} &= \frac{y_n \cdot (3 - x \cdot y_n^2)}{2}.
 \end{aligned}$$

Only two refinement steps are sufficient to achieve an accuracy of 50 bits, which we consider acceptable compared to the 53 bits accuracy of IEEE 754 double precision operations. The *rsqr*t14 instruction itself has a latency of 6 and a throughput of 2. A step of Newton's method can be carried out using 4 FMA-like instructions, so in total 9 instructions have to be executed for each inverse square root calculation. To fully occupy both scheduler ports, 8 inverse square root calculations

need to be interleaved. This on its own is possible, but for the entire force-pair calculation the amount of available registers become a limitation. Each loop iteration requires 3 registers for the inverse square root calculation and 3 are needed for holding the *diff* vector, which is used before and after the inverse square root calculation. The 3 registers needed for the inverse square root calculation may be reused for the preceding dot product, and the subsequent *force_ij* calculation. Although 6 registers would be sufficient, the GCC 8.3.0 compiler only manages to implement the loop body using 7 registers, with the extra register being used in the inverse square root calculation. Additionally, a few registers are shared among inner loop iterations: 3 registers containing the *force[i]* vector, 3 registers containing the *pos[i]* vector, 1 register containing the precalculated $-G * mass[i]$ product, and 2 registers holding the constants 3 and 0.5, which are needed for the iterative refinement. Let I be the number of interleaving loop iterations, then the number of needed registers can be calculated by

$$\#Registers = (3 + 4) \cdot I + (3 + 3 + 1 + 2) = 7 \cdot I + 10$$

There are 32 AVX registers available, meaning only 3 consecutive iterations may be interleaved. Note that manually reducing the number of registers needed per iteration from 7 to 6 does not increase the possible number of interleaved loop iterations. We have done an offline schedule of 3 interleaved iterations by hand and counted 69 cycles for this interleaved execution.

A last performance improvement can be achieved by manually interleaving loop iterations. Here, the goal is to reduce the amount of registers persistent over loop iterations by recalculating the *diff* vector after the inverse square root calculation. This way, each iteration can have its own instance of the *force[i]* vector, such that no data dependency on this vector exists and the number of persistent registers is reduced. The new number of needed registers is now given by

$$\#Registers = (3 + 3) \cdot I + (3 + 1 + 2) = 6 \cdot I + 7,$$

which allows for 4 interleaved iterations. Scheduling the 4 interleaved iterations by hand suggests that the completion of all 4 iterations takes 81 cycles. This yields the following performance for the compiled (3I) and manually interleaved (4I) implementations.

$$\begin{aligned} P_{AVX|II} &= 2 \cdot \frac{AVX_{width} \cdot II \cdot f_{max}}{T_{II}}, \\ P_{AVX512|3I} &= 2 \cdot \frac{8 \cdot 3 \cdot 2,200MHz}{69} = 1,530MPairs/s, \\ P_{AVX512|4I} &= 2 \cdot \frac{8 \cdot 4 \cdot 2,200MHz}{82} = 1,717MPairs/s. \end{aligned}$$

4.2 Parallelization

For OpenMP parallelization over multiple cores, each OpenMP thread is assigned a set of local particles for which it calculates all force interactions with all remote particles. To prevent race conditions on the accumulated forces, each thread has its own copy of a local- and remote-force array, which are accumulated together just before integration. To improve load balancing, the particles need to be assigned to the threads in a round robin scheme. Otherwise, due to the triangular loop pattern, the last thread would have more work in the order of the number of threads used than the first thread. For better cache usage, the calculation of force-pairs with remote particles is split into blocks that fit into the L1-Cache, such that all local particles p_i are used if $i < j_{max}$, where j_{max} is the maximum index of the particles in the current remote block in cache.

Similar to the FPGA implementation, different compute nodes can work together by communicating in a ring pattern, here by using MPI. Each MPI rank is assigned a subset of the particles, for which it has to calculate all force interactions with all forces in the system. At the beginning of a

Fig. 5. Performance of vectorized single core implementations. Only force calculation and accumulation.

timestep, a rank only has its own particles that then act as local and remote particles, for which it calculates all interactions acting on the local and remote forces. Afterwards, the remote particles and forces are sent to the next rank and a new set of particles is received from the preceding rank using the *MPI_Sendrecv_replace* function. This is continued until the original set of particles reaches its owner after passing through all MPI ranks once. For load balancing, the particles are again distributed in a round robin fashion. An MPI rank may own a subset of OpenMP threads, and as such the OpenMP execution was extended to support calculating the local particles of the MPI rank with the received remote particles.

5 EXPERIMENTS

All tests were conducted on dual socket Xeon Gold 6148 servers (20 cores per socket, 40 total) additionally equipped with two BittWare 520N FPGA boards. The host systems are integrated via 100 Gb/s OmniPath links into an MPI network, and as outlined in Section 3.1, each of the FPGA boards has four 40 Gb/s point-to-point links to neighboring FPGAs in a ring topology. The CPU implementation is compiled with GCC 8.3.0 and uses OpenMPI 3.1.4. The OpenCL host code for the FPGA design is compiled with GCC 7.3.0 and uses OpenMPI 3.1.1. Likwid 4.3.0 was used for profiling and power measurements on the host. Power on the FPGAs was measured with an Intel/BittWare OpenCL extension. Along with all of the performance measurements presented here, a synthetic problem setup is used that allows to quickly validate the correctness of results.

5.1 Local Performance and Problem Sizes

In the first test, we analyze the single-core performance of only the vectorized CPU force calculations described via incremental improvements in Section 4.1. Figure 5 shows the throughput in MPairs/s for different problems in multiples of 128 particles. All versions eventually get very close to their respective modeled peak performance. For the variants that use iterative refinement after the approximate *rsqrt* operation, convergence is somewhat slower, starting at around 50% of their respective peak performance for the smallest input of 128 elements compared to around 75% for the AVX2 and AVX512 baseline variants. However, despite slower convergence, the manually interleaved AVX512RSQRT4I completely dominates in absolute performance for every data point. Therefore, we proceed only with this version.

Fig. 6. Performance and power efficiency of complete force calculation and timestepping on a single node using two CPUs or two FPGAs, respectively (double precision).

For the next test, the selected force calculation loop is fully integrated into the timestepping loop and scaled to the entire compute node using the hybrid MPI + OpenMP parallelization from Section 4.2. With two MPI ranks and 20 OpenMP threads each, asymptotically, a parallel speedup of around 34 is achieved on 40 cores, as shown in the CPU part of Figure 6. Compared to the single core version, performance scaling with problem sizes behaves roughly, as expected. Extrapolating from the single core results, by scaling to 40 cores, we would expect to need $40 \cdot 128 = 5,120$ elements to reach half of the full node peak performance, which is roughly what we see for the 6,144-element test in Figure 6. The peak performance of around 60,000 MPairs/s is almost reached at around 262,144 elements, which corresponds to around 6,144 elements per core, an area where the single core implementation of Figure 5 is already essentially flat. We attribute this partially to the integration step that is now included and partially to overheads introduced by the OpenMP and MPI parallelization. Along with the performance, the power efficiency of the CPU platform increases. At medium utilization ($N = 6,144$), the absolute power consumption of two CPUs and memory is already at 273 W and increases to 324 W for twice the performance at $N = 524,288$, resulting in much higher performance/W.

The FPGA design used in comparison is synthesized from the pipeline pattern depicted in Figure 3 and configured with $16 \cdot 2$ PEs in a 4×4 layout as outlined in Section 3.3.3. Table 6 summarizes the characteristics of this design after complete technology mapping, placement, and routing through Quartus 19.4.0. In contrast to the estimates from Table 1, at this stage LUTs and FFs used are already packed together into ALMs. We also include here the utilization of block RAM, most of which is used for the local particle and force buffers. For the presented results, these kernels only use local BRAM buffers for up to 16,384 particles per device to minimize the pipeline latency. We see that corresponding to the earlier analysis, ALMs are the critical resource here. At 88% ALM usage, there is little headroom for designs with more arithmetic PEs, even though it is not evident how fully packed the ALMs are. Higher clock frequencies of up to around 400 MHz are generally possible for the combination of Stratix 10 architecture and OpenCL design entry, but the 262.5 MHz determined by the tools for this design are rather typical for such a high level of logic utilization (cf. Reference [19]). Given this clock frequency, we can now also add a model for the expected performance per FPGA for this design. It can be calculated as

$$P_{FPGA16PE,peak} = 2 \cdot 2 \cdot 16 \cdot 262.5MHz = 16,800MPairs/s,$$

Table 6. Resource Consumption of Force Calculation and Entire Design in Different Variants along with Clock Frequency for Entire Design

Region	PE Layout	Precision	ALMs	Block RAMs	DSPs	Frequency [MHz]
Force Calculation	2×2 PE Array	double	148,880	348	374	
	4×4 PE Array	double	539,274	998	1,478	
	8×8 PE Array	single	130,482	960	2,982	
Entire Design	2×2 PE Array	double	224,999	2,961	507	308.33
	4×4 PE Array	double	616,250	3,611	1,611	262.50
	8×8 PE Array	single	172,582	2,459	3,050	272.22
Available			698,450	8,953	4,713	

The force calculation consumes most logic and DSP resources (ratios depending on precision), while the remaining infrastructure uses many additional BRAM buffers.

where one of the factors 2 denotes the effect of symmetry usage as in the CPU models, the other the 2 FCUs per PE.

The FPGA measurements in Figure 6 for 2 FPGAs in one server node show that with 33,000 MPairs/s the expected performance is reached. It is actually reached already for input sizes of 1,536 particles, much earlier than for the CPU version. Thus, the FPGAs dominate performance up to a break-even point at 8,192 particles, when the higher peak performance of the CPUs take over. Since the FPGAs use less power to reach their peak performance, up to 222 W for two FPGA boards, in terms of power efficiency the FPGAs remain within 20% of the best CPU values, and the break-even is higher at 16,384 elements on one node. In addition to the more efficient on-chip communication directly streaming data from one component to the next, we attribute scaling differences to the task-level parallelism between integrator and force calculation. That is, on the FPGA the integrator never competes with the force kernel for clock cycles, as they use dedicated resources, whereas on the CPU both need to use the same resources over time.

For an additional experiment, we have investigated the impact of scaling the PE array inside the force calculation kernel on performance and problem sizes. For this, we have, on the one hand, scaled down from the 4×4 layout of PEs to a 2×2 layout without further modifications; on the other hand, we created an 8×8 layout by switching all calculations to single precision. The results in Figure 7 demonstrate that, as expected, each two-dimensional scaling step increases the performance by around 4 $\times$, up to almost 140,000 MPairs/s, with small deviations caused by the different clock frequencies. Architecturally, each of these scaling steps should also increase the problem sizes required for peak performance by 2 $\times$, which qualitatively works out with 1,536 elements for the 4×4 layout and 3,072 elements for 8×8 , but the scaling of intermediate problem sizes indicates that additional pipelining effects are in place that may encourage further investigation. Note that the 8×8 single precision goes out of the scope of our original scenario for performing all calculations in double precision and is also not fully optimized to the resource limits of the device, as demonstrated by Table 6. Hence, we proceed with the 4×4 double precision design for the further scaling experiments before we analyze the impact of floating point accuracy more closely.

5.2 Multi-node Scaling

In this section, we extend the previous experiments to a distributed multi-node setup, using the described serial channel FPGA-to-FPGA connections and MPI, respectively. First analyzing the technical merit of the parallel implementations, Figure 8 shows the weak scaling behavior of both implementations for two different problem sizes of 1,536 and 32,768 elements per node, with these

Fig. 7. Single node performance with different dimensions of PE arrays, including a 8×8 variant implemented in single precision. This variant achieves an energy efficiency of 770 MPairs/s/W @ 90 W per FPGA.

numbers chosen based on Figure 6. As the FPGA design fully overlaps communication with computation through task-level parallelism between multiple kernels, in Figure 8 it shows perfect linear scaling behavior for the tested 12 nodes and 24 FPGAs, with identical performance points for both problem sizes. The CPU implementation shows equally perfect weak scaling for the larger problem setup and only small deviations from the linear trend for the smaller one, albeit at very different performance points. Thus, the larger setup that corresponds to $12 \cdot 32,768 = 393,216$ particles on 12 nodes is around 25% faster on the CPUs than on the FPGAs, whereas the smaller one with $12 \cdot 1,536 = 18,432$ particles is almost 4× faster on the FPGAs. The weak scaling behavior itself demonstrates that both implementations are suitable to use parallel HPC resources when targeting larger problems, which is particularly important, given the quadratic scaling of computational effort. The right axis of Figure 8 additionally shows the time spent per time- or sub-step for the faster resource in each of the two scenarios. For 1,536 elements per node, the FPGA implementation allows for below 1 ms per step, even as the problem is scaled to $12 \cdot 1,536 = 18,432$ elements, whereas for 32,768 elements per node, the CPU version is faster, but simulations take in the order of 100 ms per timestep. Simulations can vary in requirements for the number of steps required as well as in the number of elements, making both scenarios relevant.

Finally, analyzing the performance in three strong scaling scenarios from 3,072 to 18,432 elements in Figure 9 further, we see that the FPGA design can achieve almost perfect scaling up to its peak performance of 400,000 MPairs/s on 12 nodes and 24 FPGAs with as little as 6,144 particles in total, and even for a total of 3,072 particles, there is still a benefit from employing 24 FPGAs. However, for these two smaller scenarios the CPU performance already degrades when using more than 2 nodes, and for $N = 18,432$ when using more than 8 nodes.

5.3 Accuracy Considerations

The N-Body system we use to test and validate the implementations is a synthetic problem designed to have an analytic solution for every number of involved particles N . These particles are constructed such that they orbit around a central point in a 2D-plane:

$$\vec{s}_i(t) = R \cdot \begin{bmatrix} \cos(\omega t + \phi_i) \\ \sin(\omega t + \phi_i) \end{bmatrix}, \phi_i = \frac{2\pi \cdot i}{N} \quad (4)$$

Fig. 8. Weak scaling of performance and time spent per step over multiple nodes. Measured time includes force calculation, accumulation, and timestepping.

Fig. 9. Strong scaling over multiple nodes. Measured time includes force calculation, accumulation, and timestepping.

$$\Rightarrow \vec{v}_i(t) = \dot{\vec{s}}_i(t) = R \cdot \omega \cdot \begin{bmatrix} -\sin(\omega t + \phi_i) \\ \cos(\omega t + \phi_i) \end{bmatrix}. \quad (5)$$

Here, $R > 0$ is the radius from the central point, ω is the angular velocity at which the particles move along the orbit, and ϕ_i is an offset for each particle, which evenly spaces them along the orbit. Setting the mass of each particle $m_i = m$ makes the construction rotationally symmetric around the central point. To simulate this behavior in a gravitational N-Body simulation, the acceleration calculated by the pairwise interactions has to match the second order derivative of the position for each particle:

$$\vec{a}_i(t) = \ddot{\vec{s}}_i(t) = -R \cdot \omega^2 \cdot \begin{bmatrix} \cos(\omega t + \phi_i) \\ \sin(\omega t + \phi_i) \end{bmatrix} \stackrel{!}{=} - \sum_{\substack{j=0 \\ i \neq j}}^{N-1} G \cdot m \cdot \frac{(\vec{s}_i - \vec{s}_j)}{|\vec{s}_i - \vec{s}_j|^3}, \quad (6)$$

Fig. 10. Mean relative error $\frac{|\vec{F}| - |\vec{F}_{ref}|}{|\vec{F}_{ref}|}$ of calculated force lengths compared to the analytical solution. Particle order was randomized before calculation to mitigate errors introduced by the uniformly constructed initial conditions.

which has to be solved for the missing parameter of the initial conditions m . One can show that,

$$m = \frac{\omega^2 R^3}{G} \div \left(\frac{1}{8} (1 + (-1)^N) + \sum_{j=1}^{\lfloor \frac{N-1}{2} \rfloor} \frac{2 + 2\cos\left(\frac{\omega}{N}(N-2j)\right)}{\left(2 - 2\cos\left(\frac{\omega \cdot j}{N}\right)\right)^{\frac{3}{2}}} \right). \quad (7)$$

This set of initial conditions has an exact analytic solution for every point in time t , but in practice it is very unstable when simulated numerically. To stabilize this setup, we introduce a central particle with a large mass $M_C = C \cdot m$, where C is the mass ratio between the central particle and any other particle in the system. In this system m can be determined similar to the unstabilized system:

$$m = \frac{\omega^2 R^3}{G} \div \left(C + \frac{1}{8} (1 + (-1)^{N-1}) + \sum_{j=1}^{\lfloor \frac{N-2}{2} \rfloor} \frac{2 + 2\cos\left(\frac{\omega}{N-1}((N-1)-2j)\right)}{\left(2 - 2\cos\left(\frac{\omega \cdot j}{N-1}\right)\right)^{\frac{3}{2}}} \right). \quad (8)$$

We use this system to demonstrate the effects of floating point accuracy on the simulation. In a first accuracy experiment, we compare the numerically calculated forces $\vec{F}$ in the initial condition with those of the analytic solution $\vec{F}_{ref}$. Figure 10 shows the relative error in the magnitude of the calculated forces. For both single and double precision, the force error increases with an increasing number of particles, but the forces calculated in double precision for a system with 131,072 elements are still more accurate than those for a 1,024-element system in single precision. Looking at concrete numbers, single precision already introduces a relative error of more than 1% for systems of only 16,384 particles. This improves when the system is made more stable by a larger central mass mitigating the accumulation errors, but the effect of arithmetic accuracy dominates, with double precision only reaching this kind of error for system sizes in the millions (exceeding Figure 10).

In a second experiment, we further analyze the effects of floating point accuracy on the full simulation, now for a small system with $N = 1,536$, where the individual force errors in single precision are still much smaller than 1%. In this experiment, we determine after how many timesteps the

Fig. 11. The orbiting system simulated until divergence for an increasing central mass. In total, two entire rotations around the central mass are simulated over 2,000 timesteps.

first particle in the system leaves its orbit, which ultimately destabilizes the system and leads to divergence from the analytical solution. This is strongly dependent on the ratio C of the central mass M_C , as it prevents close encounters by majorly contributing to the exerting force on each particle. We consider a particle divergent if its orbital radius r_i deviates from the global radius R , such that $\frac{|r_i - R|}{R} < \epsilon$, where ϵ is the relative error tolerance. Figure 11 shows the timestep when divergence occurs in relation to increasing ratios C . We see that when using double precision, the central mass can be an order of magnitude smaller than for single precision to simulate longer timescales accurately. In comparison, single precision struggles to keep the system convergent, especially for longer timescales. Further, in the scenarios in which both simulations diverge, double precision is able to simulate twice the amount of timesteps correctly.

Going beyond this limited study on the effect of single and double precision, Khrapov et al. [23] have shown that for an *all-pairs* N-Body solver, the use of single precision in a simulation of a stellar disk causes an error in the conservation quantities proportional to the amount of particles, whereas the error remains constant for double precision. From that, they claim distortions in simulation results obtained with single precision.

6 DISCUSSION AND RELATED WORK

6.1 Related Work on FPGAs

Early work for n-body problems on FPGAs focused only on the force calculations. As part of OpenDwarfs, Krommydas et al. [27] have presented an FPGA version of the n-body based calculation of electrostatic potential in GEM [11], where they achieve around 130 GFLOPS on a Xilinx Virtex-6 FPGA in single precision for the 382 atoms of Mb.HHhelix [33] with 5,884 points of interest, roughly on par with the server CPU in their comparison. Tsoi et al. [40] have performed a study of implementing a force calculation pipeline with custom number formats and precision on much earlier Xilinx Virtex-II FPGAs that required all arithmetic to be performed in logic resources. Although the concrete results are not transferable to current FPGA architectures, they underline that for a specific use case with concrete accuracy requirements, customized number formats can contribute to higher performance designs.

A more similar approach to the one presented in this article has been developed for a smaller number of previous generation Intel Arria 10 FPGAs. In their work, Huthmann et al. [15] present a distributed *all-pairs* solver in single precision without usage of force symmetry by also connecting their FPGAs in a ring. In this ring, each FPGA implements a fixed number of **processing elements (PEs)**, which form a long chained pipeline through all devices. Each PE contains a single particle for which it calculates the pair-wise interaction with all particles, which are streamed through the system. Since the number of PEs is fixed, the entire calculation is split into multiple phases, until all particles have been loaded to the PEs once. In contrast to our approach, each phase first has to initialize the PEs with particles, before the streaming computation can occur, limiting efficiency for small problem sizes.

In terms of single device implementations del Sozzo et al. [8] provide a flexible design of an *all-pairs* solver for the Xilinx line of FPGAs, specifically, the Xilinx Ultrascale+ V9UP. They are able to scale their implementation to the given device limits, by three layers of flexibility: first, a split of the input set of particles into fixed tiles of configurable size, which can be worked on independently; second, a configurable number of N-Body pipes working on the same tile, where each N-Body pipe can calculate 16 forces acting on one element in the tile at a time; and third, independent **compute units (CUs)** working on separate tiles and consisting of multiple N-Body pipes. While the device's resources can be exhausted by the design, the problem sizes necessary to achieve near peak performance are large with 40,000 particles per tile and thereby per CU. Also, their FPGA implementation only includes the force calculation, whereas the integration is done on the host CPU. With their tiling approach, this would allow for a distributed execution using host side communication via MPI, but no such parallelization is reported.

Specifically targeting MD simulations, Yang et al. [43] have presented a single-FPGA design that combines long-range force calculation via the **Particle Mesh Ewald (PME)** method and short-range forces as range-limited n-body problem. The latter is not evaluated separately from the PME pipeline, but using cells with around 80–400 particles each, it shows the general potential to combine force calculation on FPGAs with some neighbor selection scheme, which is beyond the scope of this work.

6.2 Discussion and Comparisons

In Table 7, we summarize the final force calculation performance of the demonstrated designs with regard to the problem sizes that are sufficient to reach their respective peak performance unless marked specifically. The distinction between effective and raw GFLOPs takes into account that our FPGA design uses $16 + 6 = 22$ operations per force-pair including accumulation into two sums, whereas including the iterative refinement, the CPU performs $13 + 12 + 3 + 10 = 38$ operations, 13 for the scalar force value, 12 for multiplication and accumulation of force vectors, 3 for the recomputed direction vector, and 10 in the refinement itself.

Looking first at the double precision results in Table 7, we see that in the group of results for around 384K elements, corresponding to the largest setup of our weak scaling experiment in Figure 8, our multi-CPU implementation provides the highest throughput. However, in the two result groups around 3K or 18K elements, which focus on strong scaling behavior, the presented FPGA design dominates. These strong scaling characteristics of the FPGA design depend on the perfect and fine-granular overlapping of all phases of computation and data movement and thus is partially enabled by the streaming communication interface. While alternative communication via PCIe and the MPI on the host would certainly involve large overheads (see Reference [36] for measurements on individual data transfer latencies), it would be interesting to evaluate a block-based direct FPGA-to-FPGA communication infrastructure like that introduced by Kobayashi et al. [25] in future work.

Table 7. Summary and Comparison of Final Performance Metrics of Presented Designs and Related Work

Design	Platform	Precision	N	effective GPairs/s	effective GFLOPS	raw GFLOPS	theor. peak GFLOPS	power [W]
this work	2 Stratix 10 FPGAs	double	1,536	33	363	363		220
this work	2 Stratix 10 FPGAs	single	3,072	140	1,540	1,540		180
[15]	2 Arria 10 FPGAs	single	1,024*	0.25				
[35]	1 Xeon 6148 CPU	mixed	3,000	30†			2,816 FP32	
[29]	1 GRAPE-8 ASIC board	custom	2,000*			110	960	46
this work	24 Stratix 10 FPGAs	double	18K	400	4,400	4,400		2,670
[15]	8 Arria 10 FPGAs	single	32K*	30				
this work	16 Xeon 6148 CPUs	double	18K*	120	1,320	2,280	22,528 FP64	2,080
[5]	4 C2050 GPUs	mixed	16K*			750	4,120 FP32	
[29]	1 GRAPE-8 ASIC board	custom	20K*			550	960	46
this work	24 Stratix 10 FPGAs	double	384K	400	4,400	4,400		2,670
[8]	1 Xilinx VU9P FPGA	single	360K	13.4				20
[15]	8 Arria 10 FPGAs	single	512K	47				
this work	24 Xeon 6148 CPUs	double	384K*	506	5,566	9,614	33,792 FP64	3,960
this work	2 Xeon 6148 CPUs	double	262K	60	660	1,140	2,816 FP64	318
[8]	2 Xeon E5-2680 v2 CPUs	single	360K	2.6			896 FP32	
[5]	4 C2050 GPUs	mixed	228K			2,000	4,120 FP32	
[35]	1 V100 GPUs	mixed	192K	167†			14,000 FP32	
this work	24 Xeon 6148 CPUs	double	1.8M	630	6,930	11,970	33,792 FP64	3,960
[5]	1536 C2050 GPUs	mixed	6M			350,000	1,582,080 FP32	

Precision as single (FP32), double (FP64), mixed (FP32 force calculation, FP64 accumulation), and custom (FP18 force calculation, 64-bit fixed point accumulation), respectively. Measurements grouped by problem sizes, where asterisk* marks problem sizes that cause execution below peak performance due to insufficient problem sizes. † Effective GPairs/s values from Reference [35] are not directly comparable as they are correspond to raw GPairs/s based on their Figure 6 (assuming reported pairs are used twice) with size scaling derived from their Figure 7. Neighbor lists reduce the effective GPairs/s depending on radius and particle density. For our work, the iterative refinement for *rsqrt* leads to a distinction between effective and raw GFLOPS for the CPU version. Raw GFLOPS values for Reference [5] derived from their Figure 5. Theoretical peak performance for given precision derived from CPU and GPU specifications. FPGA peak performance is hard to quantify due to the large gap between practical clock frequencies and device specifications. Best CPU and GPU designs reach close to 50% of their theoretical peak. Reported power values are only those measured, not the TDP.

In addition to diligent optimization efforts, the relation between the demonstrated CPU and FPGA results depends on several conditions. First, the slow special function arithmetic for inverse square root calculations had almost restricted the CPU throughput to half of the final throughput (Figure 5) that was only be recovered by a special workaround. More or different special arithmetic operations could easily break the involved balance of vector operations, whereas on FPGA, the special arithmetic is just generated as needed at moderate resource cost. Second, only the manual scheduling of CPU vector instructions from interleaved loop iterations has mitigated the still present performance bottleneck caused by shortage of vector registers. The FPGA, however, provides sufficient registers along the entire data path even for larger DFGs than the n-body forces in Figure 1. And third, a switch from double to single precision arithmetic levels the playing field. The CPU design, not specifically ported to single precision, should see around 2× increase in raw performance, with some additional gain possible for the limitations around inverse square roots. However, with the same architecture approach, the FPGA design flips from the observed logic limitation to a DSP limited design point. In Section 5.1, we have demonstrated a 4× speedup for a pure single precision design, with possible headroom for up to 6× speedup based on DSP utilization, in line to throughput indicated by other arithmetic limited types of FPGA designs [12, 19]. Comparing the single precision design to the design by Huthmann [15] reveals our algorithmic

and architectural improvements. For this, we consider their performance projection to the newer Intel Agilex AGI027 of 20 GPairs/s per FPGA. This offset to our 70 GPairs/s per device can be explained by, on the one hand, the $2\times$ algorithmic performance improvement on our side and, on the other hand, by their ALM limited design, which seems to be unable to utilize the higher DSP count of the Intel Agilex AGI027. Second, the architectural differences result in their design performing much worse for smaller problem sizes. Moving on to the work of del Sozzo [8], a fair comparison in terms of peak performance is more difficult, due to the architectural differences in Xilinx and Intel FPGAs. Even though the devices used for the experiments are comparable in terms of logic elements, the Intel DSPs offer hardened single precision multipliers and adders, ultimately giving them the edge in floating point applications. Considering this, it is interesting to note that given the 20W power usage of the VU9P board reported in Reference [8], their single precision energy efficiency seems very close to that of our single precision design. However, their design based on tiling with insufficient overlapping of phases is not well suited for small problem sizes, which is independent of the floating point implementation.

Moving to mixed- or fully custom-precision designs can reveal further potential for the FPGA architecture. This has already proven useful for ASIC designs such as Grape-8 [29]. On the much older 90 nm technology, a Grape-8 card has a peak performance of 1TFlop/s, which is comparable to the performance of modern devices, albeit it is using a custom 18-bit number format. To make use of the arithmetic potential of GPUs, codes such as NBODY6++GPU [41] and ϕ -GPU [4] have moved to mixed-precision implementations with force calculation in single and accumulation in double precision. We have included their results [5] on previous generation GPUs in Table 7, as well as recent GROMACS results [35] that use a current GPU and the same CPU as in our experiments. The GROMACS metrics are not fully comparable, as they perform local force calculations with neighbor lists. As best-case scenario, we take their raw MPairs/s metric and assume each reported interaction is used twice. The results show that in mixed precision, GPUs can achieve high performance for medium-sized problems (here, 192K elements) on few devices, with the highest absolute performance of our comparison also achieved by GPUs for millions of particles, albeit on a much larger cluster than available for all other experiments.

The presented FPGA design can be adapted to other devices in the Stratix10 family. Table 8 demonstrates how the local- and remote dimension of the PE array may be adjusted to account for different amounts resources in respective Stratix10 devices. A second level of flexibility is the number of particles each FPGA can hold locally. For the presented designs, this was set to 32,768, but this can be scaled down in powers of two to suit the number of Block RAMs of the given device.

7 CONCLUSION AND OUTLOOK

In the current setup, the particular value proposition of the presented FPGA implementation is its strong scaling behavior, which allows it to provide highest absolute performance for problem sizes where other architectures do not reach their respective peak performance yet. For the concrete example of 18,432 particles, the multi-FPGA setup is the only one that achieves sub-millisecond step times, allowing for long simulation time scales at 1,177 timesteps or leap-frog sub-steps per second. In case this scaling behavior persists for larger FPGA clusters with hundreds or thousands of devices—which seems plausible given the architecture, but remains hypothetical at this point—FPGAs could also become the fastest architecture for a problem size of 192K elements on 240 FPGAs or 1.8M elements on 2,400 FPGAs. Except for the Microsoft Azure infrastructure [6, 10], which is not open for this type of research, such infrastructure has yet to be built.

As one use case for building FPGA-accelerated HPC systems of that scale, it would help to extend the presented architecture to the feature set of full scientific simulation software. Beyond the adaptive timesteps prevalent in cosmological simulations with new challenges involving neighbor

Table 8. Potentially Smallest-fitting FPGA Devices for Different Variants of the Presented Design

FPGA Product Specification	Design Synthesis Result	absolute resources			relative resources		
		ALMs	Block RAMs	DSPs	ALMs	BRAMs	DSPs
Stratix10 GX 650	1 × IPE Array	127,057	2,670	207	61.2%	107.2%	17.9%
		207,360	2,489	1,152			
Stratix10 GX 850	2 × IPE Array	165,796	2,773	299	58.1%	79.7%	14.8%
		284,960	3,477	2,016			
Stratix10 GX 1100	2 × 2PE Array	233,844	2,950	483	52.0%	54.0%	18.6%
		449,280	5,461	2,592			
Stratix10 GX 1650	4 × 2PE Array	376,482	3,178	851	68.4%	54.3%	27.1%
		550,540	5,851	3,145			
Stratix10 GX 2500	4 × 4PE Array	649,351	3,600	1,587	79.1%	36.1%	31.7%
		821,150	9,963	5,011			

Synthesis results after place and route obtained on reference Stratix 10 GX2800. FPGA devices from product table were chosen, such that the design leaves at least 20% headroom for the static partition needed for memory external interface and communication. The presented synthesis results are all configured for 32,768 particles per device, such that observed differences in Block RAM usage reflect changes in the PE array; number of particles could be adapted to resolve Block RAM limitations of smaller devices.

lists, future algorithmic improvements could include a transition to Hermite integrators [14, 28]. In addition to the acceleration (force) term, this would also involve the calculation the jerk J (first order derivative of acceleration) of every particle. Since the jerk term itself still satisfies the condition $J_{ij} = J_{ji}$, the presented architecture is suitable for such an extension, but the balance of resources would change. Architecturally interesting extensions could also involve heterogeneous simulations using FPGA + CPU or FPGA + GPU resources for different tasks, possible along with a generalization to a different communication infrastructure that may connect these devices not as accelerator and host, but as communication peers.

REFERENCES

- [1] Intel. 2020. Intel® 64 and IA-32 Architectures Optimization Reference Manual. (May 2020).
- [2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report. University of California at Berkeley.
- [3] Josh Barnes and Piet Hut. 1986. A hierarchical $O(N \log N)$ force-calculation algorithm. *Lett. Nat.* 324 (1986), 446–449.
- [4] Peter Berczik, Keigo Nitadori, Shiyan Zhong, Rainer Spurzem, Tsuyoshi Hamada, Xiaowei Wang, Ingo Berentzen, Alexander Veles, and Wei Ge. 2011. High performance massively parallel direct N-body simulations on large GPU clusters. In *Proceedings of the International Conference on High-performance Computing*. 8–18.
- [5] Peter Berczik, Rainer Spurzem, Shiyan Zhong, Long Wang, Keigo Nitadori, Tsuyoshi Hamada, and Alexander Veles. 2013. Up to 700k GPU cores, kepler, and the exascale future for simulations of star clusters around black holes. In *Proceedings of the 28th International Supercomputing Conference (Lecture Notes in Computer Science)*, Julian M. Kunkel, Thomas Ludwig, and Hans Werner Meuer (Eds.), Vol. 7905. Springer, 13–25. DOI: https://doi.org/10.1007/978-3-642-38750-0_2
- [6] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. 2016. A cloud-scale acceleration architecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 1–13. DOI: <https://doi.org/10.1109/MICRO.2016.7783710>
- [7] Tom Darden, Darrin York, and Lee Pedersen. 1993. Particle mesh Ewald: An $N \cdot \log(N)$ method for Ewald sums in large systems. *J. Chem. Phys.* 98, 12 (1993), 10089–10092. DOI: <https://doi.org/10.1063/1.464397>
- [8] Emanuele Del Sozzo, Marco Rabozzi, Lorenzo Di Tucci, Donatella Sciuto, and Marco D. Santambrogio. 2018. A scalable FPGA design for cloud N-body simulation. In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*. 1–8. DOI: [10.1109/ASAP.2018.8445106](https://doi.org/10.1109/ASAP.2018.8445106)

- [9] Ulrich Essmann, Lalith Perera, Max L. Berkowitz, Tom Darden, Hsing Lee, and Lee G. Pedersen. 1995. A smooth particle mesh Ewald method. *J. Chem. Phys.* 103, 19 (1995), 8577–8593. DOI : <https://doi.org/10.1063/1.470117>
- [10] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 51–66. Retrieved from <https://www.usenix.org/conference/nsdi18/presentation/firestone>.
- [11] John C. Gordon, Andrew T. Fenley, and Alexey Onufriev. 2008. An analytical approach to computing biomolecular electrostatic potential. II. Validation and applications. *J. Chem. Phys.* 129, 075102 (August 2008). DOI : <https://doi.org/10.1063/1.2956499>
- [12] P. Gorlani, T. Kenter, and C. Plessl. 2019. OpenCL implementation of cannon’s matrix multiplication algorithm on Intel Stratix 10 FPGAs. In *Proceedings of the International Conference on Field-programmable Technology (ICFPT)*. 99–107. DOI : <https://doi.org/10.1109/ICFPT47387.2019.00020>
- [13] L. Greengard and V. Rokhlin. 1987. A fast algorithm for particle simulations. *J. Comput. Phys.* 73, 2 (1987), 325–348. DOI : [https://doi.org/10.1016/0021-9991\(87\)90140-9](https://doi.org/10.1016/0021-9991(87)90140-9)
- [14] Piet Hut and Jun Makino. 2007. The 2-Body Problem: Higher-order Integrators. Retrieved from http://www.artcompsci.org/kali/vol/two_body_problem_2/ch07.html.
- [15] Jens Huthmann, Abiko Shinand Artur Podobas, Kentaro Sano, and Hiroyuki Takizawa. [n.d.]. Scaling performance for N-body stream computation with a ring of FPGAs. In *Proceedings of the International Symposium Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*. IEEE, 1–6.
- [16] Intel. 2020. Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide (UG-OCL003 | 2020.12.14, Version 20.4). Retrieved from <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl-best-practices-guide-20-4.pdf>.
- [17] Intel. 2020. Intel Stratix 10 Variable Precision DSP Blocks User Guide (UG-S10-DSP | 2020.09.28, Version 20.3). Retrieved from <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10-ug-s10-dsp.pdf>.
- [18] A. K. Jain, H. Omidian, H. Fraisse, M. Benipal, L. Liu, and D. Gaitonde. 2020. A domain-specific architecture for accelerating sparse matrix vector multiplication on FPGAs. In *Proceedings of the International Conference on Field-programmable Logic and Applications (FPL)*. 127–132. DOI : <https://doi.org/10.1109/FPL50879.2020.00031>
- [19] Martin Karp, Artur Podobas, Niclas Jansson, Tobias Kenter, Christian Plessl, Philipp Schlatter, and Stefano Markidis. 2021. High-performance spectral element methods on field-programmable gate arrays. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*. 1077–1086. DOI : <https://doi.org/10.1109/IPDPS49936.2021.00116>
- [20] Martin Karp, Artur Podobas, Tobias Kenter, Niclas Jansson, Christian Plessl, Philipp Schlatter, and Stefano Markidis. 2022. A high-fidelity flow solver for unstructured meshes on field-programmable gate arrays: Design, evaluation, and future challenges. In *Proceedings of the International Conference on High-performance Computing in Asia-Pacific Region*. DOI : [10.1145/3492805.3492808](https://doi.org/10.1145/3492805.3492808)
- [21] Tobias Kenter, Jens Förstner, and Christian Plessl. 2017. Flexible FPGA design for FDTD using OpenCL. In *Proceedings of the International Conference on Field-programmable Logic and Applications (FPL)*. IEEE, 1–7. DOI : <https://doi.org/10.23919/FPL.2017.8056844>
- [22] Tobias Kenter, Adesh Shambhu, Sara Faghih-Naini, and Vadym Aizinger. 2021. Algorithm-hardware co-design of a discontinuous Galerkin shallow-water model for a dataflow architecture on FPGA. In *Proc. Platform for Advanced Scientific Computing Conf. (PASC)*. DOI : <https://doi.org/10.1145/3468267.3470617>
- [23] S. S. Khrapov, S. A. Khoperskov, and A. V. Khoperskov. 2018. New features of parallel implementation of N-body problems on GPU. *Bull. South Ural State Univ., Series: Math. Model., Program. Comput. Softw.* 1 (2018), 124–136. DOI : <https://doi.org/10.14529/mmp180111>
- [24] Hiroshi Kinoshita, Haruo Yoshida, and Hiroshi Nakai. 1990. Symplectic integrators and their application to dynamical astronomy. *Celest. Mech. Dynam. Astron.* 50 (1990), 59–71. DOI : <https://doi.org/10.1007/BF00048986>
- [25] Ryohei Kobayashi, Yuma Oobata, Norihisa Fujita, Yoshiki Yamaguchi, and Taisuke Boku. 2018. OpenCL-ready high speed FPGA network for reconfigurable high performance computing. In *Proceedings of the International Conference on High-performance Computing in Asia-Pacific Region (HPC Asia’18)*. Association for Computing Machinery, New York, NY, 192–201.
- [26] Bartosz Kohnke, Carsten Kutzner, and Helmut Grubmüller. 2020. A GPU-accelerated fast multipole method for GRO-MACS: Performance and accuracy. *J. Chem. Theor. Comput.* 16, 11 (Oct. 2020), 6938–6949. DOI : <https://doi.org/10.1021/acs.jctc.0c00744>

- [27] Konstantinos Krommydas, Wu-chun Feng, Christos D. Antonopoulos, and Nikolaos Bellas. 2015. OpenDwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *J. Sig. Process. Syst.* 85, 3 (2015), 1–20. DOI : <https://doi.org/10.1007/s11265-015-1051-z>
- [28] Junichiro Makino and Sverre J. Aarseth. 1992. On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems. *Publicat. Astron. Societ. Japan* 44 (1992), 141–151.
- [29] Junichiro Makino and Hiroshi Daisaka. 2012. GRAPE-8—An accelerator for gravitational N-body simulation with 20.5Gflops/W performance. In *Proceedings of the International Conference on High-performance Computing, Networking, Storage and Analysis (SC)*. 1–10.
- [30] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefer. 2019. FBLAS: Streaming linear algebra on FPGA. *CoRR abs/1907.07929* (2019).
- [31] Shoichi Oshino, Yoko Funato, and Junichiro Makino. 2011. Particle–particle particle–tree: A direct-tree hybrid scheme for Collisional N-Body simulations. *Publicat. Astron. Societ. Japan* 63, 4 (Aug. 2011), 881–892. DOI : <https://doi.org/10.1093/pasj/63.4.881>
- [32] Peter S. Pacheco. 2011. *An Introduction to Parallel Programming*. Morgan Kaufmann, Burlington, MA, 292–299.
- [33] S. E. Phillips. 1980. Structure and refinement of oxymyoglobin at 1.6 Å resolution. *J. Molec. Biol.* 142, 4 (Oct. 1980), 531–554. DOI : [https://doi.org/10.1016/0022-2836\(80\)90262-4](https://doi.org/10.1016/0022-2836(80)90262-4)
- [34] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
- [35] Szilárd Páll, Artem Zhmurov, Paul Bauer, Mark Abraham, Magnus Lundborg, Alan Gray, Berk Hess, and Erik Lindahl. 2020. Heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS. *J. Chem. Phys.* 153, 134110 (2020), 1–15. DOI : <https://doi.org/10.1063/5.0018516>
- [36] Arjun Ramaswami, Tobias Kenter, Thomas D. Kühne, and Christian Plessl. 2021. Evaluating the design space for offloading 3D FFT calculations to an FPGA for high-performance computing. *Appl. Reconfig. Comput. Archit. Tools Applic.* 12700 (2021), 285–294.
- [37] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. 2014. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Trans. Parallel Distrib. Syst.* 25, 3 (Mar. 2014), 695–705. DOI : <https://doi.org/10.1109/TPDS.2013.51>
- [38] Volker Springel. 2005. The cosmological simulation code GADGET-2. *Month. Not. Roy. Astron. Societ.* 364 (2005).
- [39] Lawrence C. Stewart, Carlo Pasoe, Brian W. Sherman, Martin Herbordt, and Vipin Sachdeva. 2020. An OpenCL 3D FFT for molecular dynamics simulations on multiple FPGAs. *arXiv preprint arXiv:2009.12617* (2020).
- [40] K. H. Tsoi, C. H. Ho, H. C. Yeung, and P. H. W. Leong. 2004. An arithmetic library and its application to the N-body problem. In *Proceedings of the IEEE Symposium on Field-programmable Custom Computing Machines (FCCM'04)*. IEEE Computer Society, 68–78. DOI : <https://doi.org/10.1109/FCCM.2004.14>
- [41] Long Wang, Rainer Spurzem, Sverre Aarseth, Keigo Nitadori, Peter Berczik, M. B. N. Kouwenhoven, and Thorsten Naab. 2015. NBODY6++GPU: Ready for the gravitational million-body problem. *Month. Not. Roy. Astron. Societ.* 450, 4 (May 2015), 4070–4080. DOI : <https://doi.org/10.1093/mnras/stv817>
- [42] Mathias Winkel, Robert Speck, Helge Hübner, Lukas Arnold, Rolf Krause, and Paul Gibbon. 2012. A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale N-body simulations. *Comput. Phys. Commun.* 183, 4 (2012), 880–889. DOI : <https://doi.org/10.1016/j.cpc.2011.12.013>
- [43] Chen Yang, Tong Geng, Tianqi Wang, Rushi Patel, Qingqing Xiong, Ahmed Sanaullah, Chunshu Wu, Jiayi Sheng, Charles Lin, Vipin Sachdeva, Woody Sherman, and Martin Herbordt. 2019. Fully integrated FPGA molecular dynamics simulations. In *Proceedings of the International Conference on High-performance Computing, Networking, Storage and Analysis (SC)*. 1–31.
- [44] Haruo Yoshida. 1990. Construction of higher order symplectic integrators. *Phys. Lett. A* 150 (1990), 262–268.
- [45] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *Proceedings of the International Symposium on Field-programmable Gate Arrays (FPGA'18)*. ACM, New York, NY, 153–162. DOI : <https://doi.org/10.1145/3174243.3174248>

Received June 2021; revised September 2021; accepted October 2021