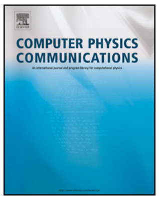
Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc



Development and performance of a HemeLB GPU code for human-scale blood flow simulation [☆,☆☆]

I. Zacharoudiou [a], J.W.S. McCullough [a], P.V. Coveney [a,b,*]

[a] *Centre for Computational Science, Department of Chemistry, University College London, UK*
[b] *Informatics Institute, University of Amsterdam, Netherlands*

A R T I C L E   I N F O

Article history:
Received 5 December 2021
Received in revised form 28 August 2022
Accepted 15 September 2022
Available online 22 September 2022

Keywords:
Blood flow modelling
Lattice Boltzmann method
High performance computing

A B S T R A C T

In recent years, it has become increasingly common for high performance computers (HPC) to possess some level of heterogeneous architecture - typically in the form of GPU accelerators. In some machines these are isolated within a dedicated partition, whilst in others they are integral to all compute nodes - often with multiple GPUs per node - and provide the majority of a machine's compute performance. In light of this trend, it is becoming essential that codes deployed on HPC are updated to execute on accelerator hardware. In this paper we introduce a GPU implementation of the 3D blood flow simulation code HemeLB that has been developed using CUDA C++. We demonstrate how taking advantage of NVIDIA GPU hardware can achieve significant performance improvements compared to the equivalent CPU only code on which it has been built whilst retaining the excellent strong scaling characteristics that have been repeatedly demonstrated by the CPU version. With HPC positioned on the brink of the exascale era, we use HemeLB as a motivation to provide a discussion on some of the challenges that many users will face when deploying their own applications on upcoming exascale machines.

Program summary
Program Title: HemeLB (HemePure_GPU)
CPC Library link to program files: https://doi.org/10.17632/jhdv4drxbx.1
Developer's repository link: https://github.com/UCL-CCS/HemePure-GPU
Licensing provisions: LGPLv3
Programming language: CUDA C++
Nature of problem: The geometric characteristics of arteries and veins throughout the human body can vary considerably between individuals. This is particularly true for patients with cardiovascular disease. Accurately resolving the dynamics of blood flow within such domains requires a three-dimensional technique that can replicate such variations at high fidelity. The study of full human-scale domains for physiologically relevant timeframes further demands a tool that can be executed efficiently on the advancing technological frameworks available on modern high performance computers.
Solution method: HemeLB uses the lattice Boltzmann method [1,2,3,4] to simulate blood flow in complex, three-dimensional sparse vasculatures. A single relaxation time approximation is used [5]. Solid boundaries are modelled using simple bounce-back boundary conditions [6]. Blood flow is driven by applying either velocity or pressure boundary conditions [7]. The localised solution kernels of the algorithm allow for efficient parallelisation of the method to very high core counts. This version of HemeLB outlines the conversion of the code to allow execution on NVIDIA GPUs, currently the most widely used architecture in ultra high performance computers, whilst retaining its capacity for strong scaling to very large core counts.

References

[1] S. Succi, The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond, Oxford University Press, 2001.
[2] A.A. Mohamad, Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes, Springer London, 2011.

☆ The review of this paper was arranged by Prof. David W. Walker.
☆☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding author at: Centre for Computational Science, Department of Chemistry, University College London, UK.
E-mail address: p.v.coveney@ucl.ac.uk (P.V. Coveney).

https://doi.org/10.1016/j.cpc.2022.108548

[3] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E.M. Viggen, The Lattice Boltzmann Method: Principles and Practice, Springer, 2017.
[4] S. Succi, The Lattice Boltzmann Equation: For Complex States of Flowing Matter, Oxford University Press, Oxford, 2018.
[5] P.L. Bhatnagar, E.P. Gross, and M. Krook, Physical review, 94 (3) (1954) 511.
[6] A.J.C. Ladd and R. Verberg, Journal of statistical physics, 104 (5) (2001) 1191–1251.
[7] R.W. Nash, H.B. Carver, M.O. Bernabeu, J. Hetherington, D.K. Groen, T. Krüger, and P.V. Coveney, Phys. Rev. E, 89 (2014) 023303.

1. Introduction

In the field of computational biomedicine, significant effort is being invested into the development of the virtual human - a digital twin of an individual's physiology. A full virtual human would allow clinicians, scientists and healthcare professionals to make use of patient-specific simulations and predictive models to optimise the care provided to an individual at all stages of life. Conducting simulations on virtual humans will contribute in a major way to our understanding of the several body systems and processes involved, such as the cardiovascular system and the circulatory physiology.

Several techniques and models have been developed in this direction, with varying dimensionality; from zero to three dimensional models, each category has its limitations and range of application [1]. Zero-dimensional or lumped parameter models provide a way of evaluating the hemodynamic interactions among the cardiovascular organs. These models, however, ignore any spatial variation of the fundamental variables, such as pressure and flow, and assume a uniform distribution within the system at a given time. Higher dimensional models on the other hand take into account the spatial variation of these parameters.

This is important, for example, for simulating the blood flow through vascular networks, which has been conducted widely using both 1D and 3D solvers [2–8]. Both approaches have advantages and disadvantages relating to implementation, resolution of output, assumptions made, and solution time. For example, 1D models are far simpler to implement and faster to solve, especially as networks get larger; however, the approach makes significant assumptions about the geometric properties of a vessel and the detail of flow patterns obtained is limited. On the other hand, 3D models can generate highly detailed flow fields in patient-specific representations of vessels but do so with greater computational effort and significantly increased quantities of data in/output. A combination of methods is also feasible or desired in some cases. In multi-scale models, low dimensional models are increasingly used to provide the boundary conditions for the more sophisticated, complex and potentially patient-specific 3D models [1].

The increasing performance, capacity and availability of computing architectures means that 3D models of large and detailed geometries can be studied in a tractable time frame. This is further enhanced when the 3D code scales efficiently up to large core counts. This is extremely useful and of paramount importance, as conducting simulations on virtual humans will demand codes that can both execute and scale efficiently on large-scale supercomputers.

The purpose of this paper is to provide another step forward in the effort to make the virtual human a reality by taking advantage of the accelerators such as Graphics Processing Units (GPUs) that are becoming commonplace on supercomputers globally. We do this by developing a GPU version of HemeLB [9,10], a high-performance lattice Boltzmann (LB) based fluid flow solver for simulating blood flow on patient specific images obtained from medical scans. HemeLB has been optimized for the sparse geometries characteristic of vascular trees and has demonstrated strong scaling on hundreds of thousands of CPU cores on non-accelerated HPC including Blue Waters and SuperMUC-NG. Our vision for this work is to demonstrate HemeLB's capacity for execution on the largest and fastest current generation machines and prepare it for upcoming exascale machines. In both of these settings, a considerable part of the performance is sourced from the presence of accelerators on the nodes, typically GPUs. Here we present our implementation of a GPU version of HemeLB and report performance and scaling analysis up to tens of thousands of GPUs.

The paper is organised as follows. The remainder of this introductory section is devoted to related work on modelling aspects and the lattice Boltzmann method. Section 2 briefly presents the details of the lattice Boltzmann method used for simulating 3D blood flow through vascular geometries. Section 3 is devoted to the GPU code implementation. Large scale performance comparison of the CPU and GPU versions of HemeLB is carried out in section 4 focusing on their strong scaling performance; we also examine which metrics should be used in assessing this performance, given the inherent differences between CPUs and GPUs and how HPC platforms are configured. Section 5 examines challenges related to running massively parallel simulations at the emerging exascale, while section 6 presents a simple example of simulating blood flow using HemeLB_GPU. Finally, future work plans and conclusions drawn from this work are reported in sections 7 & 8 respectively.

1.1. Background - related work

HemeLB is a LB fluid flow solver for simulating blood flow. The lattice Boltzmann method (LBM) is an alternative to classical Computational Fluid Dynamics (CFD) methods for numerically solving the Navier-Stokes equations. It is increasingly gaining interest in various scientific areas, due to its simplicity, scalability on parallel computers, ease of handling complex and sparse geometries, as well as incorporating multi-scale, multi-physics phenomena.

Various LB based, high performance codes, have been developed by several research groups, including *Palabos* [11], *waLBerla* [12,13], *MUPHY* [14,15], *HARVEY* [7,16] and *stlbm* [17]. These codes target different types of hardware, either many-core CPU platforms (*Palabos*, *HARVEY*), or heterogeneous many-core CPU/GPU platforms (*Palabos*, *MUPHY*, *waLBerla*). A hardware-agnostic implementation strategy for LB simulations on homogeneous and heterogeneous many-core systems has been recently proposed by Latt et al. [17] based solely on C++17 Parallel Algorithms.

LBM is well suited for acceleration on GPUs, which use the single instruction, multiple data (SIMD) paradigm, as LBM applies the same set of instructions on all fluid sites. A significant amount of work discusses GPU implementations of the LBM, both in dense geometries, where the majority of the lattice points are fluid sites [15,18–21], but also in complex, sparse geometries, like blood flow in vascular geometries [22–25] or flow in porous media [26,27].

As LB based codes are heavily data intensive and memory bound algorithms, significant effort has been devoted on data storage schemes and memory access patterns [17,20,25,28–31]. There are three issues regarding data storage within the LBM framework: (i) how the three-dimensional lattice points are ordered in a one-dimensional array, (ii) how the particle distribution functions (at the heart of LBM, see section 2.2) are stored and accessed from memory and (iii) how the geometric characteristics of the simulation domain are preserved in a way that minimizes memory requirements, which is extremely important for sparse geometries like vascular domains. The above issues are denoted as "memory ordering", "memory/data layout" and "memory addressing" respectively. A discussion on these different data storage schemes for GPUs is provided in [30]; a comparison of some of these approaches on various platforms, homogeneous and heterogeneous, has been carried out by Latt et al. [17], highlighting the code's performance dependency on the memory access patterns and the architecture of the hardware used.

Coupled schemes have been also developed. Feiger et al. [32] used a combination of machine learning and massively CPU parallel LB fluid simulations using *HARVEY* [7] to predict the effects of physiological factors on hemodynamics in patients with coarctation of the aorta. Massive coupled LB-MD multi-GPU simulations, studying proteins in crowding conditions, were carried out by Bernaschi et al. [15], demonstrating excellent scalability up to 18 000 K20X Nvidia GPUs.

Challenges related to massively parallel CPU simulations of hemodynamics are discussed by Randles et al. [7]. These include the memory footprint, I/O bandwidth, scalability and load imbalance. The authors extended the capabilities of *HARVEY* [16] to address the above issues and ran LB simulations on 1,572,864 CPU cores of the IBM Blue Gene/Q (Lawrence Livermore National Laboratory (LLNL)). Load imbalance was the major factor identified for the deviation from ideal strong scaling observed. We will discuss further challenges at the emerging exascale in section 5.

2. Numerical methods

In this paper we use the LBM to solve 3D blood flow through a vascular geometry. A brief outline of this approach is presented in this section.

2.1. Equations of motion

Modelling 3D blood flow is a problem of computational fluid dynamics. The blood consists primarily of blood cells suspended in blood plasma. From a rheological perspective, blood plasma has been considered a Newtonian fluid, while whole blood behaves as a non-Newtonian fluid; whole blood, however, follows Newtonian behaviour when the shear rate exceeds 100 s^{-1}. Therefore, for flows in large blood vessels, e.g. aorta or large arteries, the effect of the non-Newtonian nature of blood is not significant and considering the flow as Newtonian is a satisfactory assumption.

The hydrodynamic incompressible equations of motion that describe the blood flow are the continuity, eq. (1), and the Navier-Stokes, eq. (2), equations [1,6,33]

$$\partial_t \rho + \partial_\alpha(\rho u_\alpha) = 0\,, \tag{1}$$

$$\partial_t(\rho u_\alpha) + \partial_\beta(\rho u_\alpha u_\beta) = -\partial_\alpha p + \partial_\beta\left[\eta\left(\partial_\beta u_\alpha + \partial_\alpha u_\beta\right)\right]\,, \tag{2}$$

where $\mathbf{u}$, ρ, p, η are the fluid velocity, density, pressure and dynamic viscosity respectively.

2.2. The lattice Boltzmann method

It is not the purpose of this paper to give a full and detailed description of the LBM, as this is widely available in the literature e.g. [34–38]. We will, however, summarise the key features of our implementation.

To solve the hydrodynamic equations, eqs. (1), (2), with the LBM, the domain is partitioned into a Cartesian grid with a constant spacing of Δx in all 3D directions. At each nodal location, $\mathbf{x}$, a discrete set of distribution functions, $f_i(\mathbf{x},t)$, is assigned to represent the amount of fluid moving in direction i at time t. In this work, we use a three-dimensional model with 19 discrete velocity vectors (D3Q19), where fluid can stay at the current location or move to one of the 18 neighbours described by the sets: $i = 1-6$ $[(\pm1,0,0), (0,\pm1,0), (0,0,\pm1))$ and $i = 7-18$ $[(\pm1,\pm1,0), (\pm1,0,\pm1), (0,\pm1,\pm1)]$. The flow described by $f_i(\mathbf{x},t)$ evolves over the time step Δt

$$\text{Collision step:} \qquad f_i'(\mathbf{x},t) = f_i(\mathbf{x},t) - \frac{\Delta t}{\tau}\left(f_i(\mathbf{x},t) - f_i^{eq}(\mathbf{x},t)\right)\,, \tag{3a}$$

$$\text{Streaming step:} \qquad f_i(\mathbf{x}+\mathbf{c_i}\Delta t, t+\Delta t) = f_i'(\mathbf{x},t)\,. \tag{3b}$$

Eq. 3 states that the time evolution of the distribution functions proceeds in two steps: (a) a collision step with f_i relaxing towards their equilibrium state f_i^{eq} (Maxwell-Boltzmann distribution) over a timescale τ, within a single relaxation time approximation (BGK collision kernel) [39], giving locally the updated or post-collisional distribution functions f_i' and (b) a streaming step with velocity $\mathbf{c}_i$ to the neighbouring lattice point $\mathbf{x}+\mathbf{c_i}\Delta t$ at the next time step $t+\Delta t$.

The equilibrium distributions functions are defined as a power series in the velocity

$$f_i^{eq}(\mathbf{x},t) = w_i\rho(\mathbf{x},t)\left(1 + \frac{\mathbf{c}_i\cdot\mathbf{u}}{C_s^2} + \frac{(\mathbf{c}_i\cdot\mathbf{u})^2}{C_s^4} - \frac{|\mathbf{u}|^2}{C_s^2}\right), \tag{4}$$

with the coefficients, w_i, for the D3Q19 model, being 1/3 for $i=0$ (the source node), 1/18 for $i=1-6$ and 1/36 for $i=7-18$. C_s represents the speed of sound of the fluid and evaluates to $\frac{1}{\sqrt{3}}$.

A Chapmann-Enskog expansion [40] can be used to demonstrate that this framework leads to the hydrodynamic equations, continuity eq. (1) and Navier-Stokes eqs. (2) in a low Mach number limit.

Local macroscopic properties of density and momentum can be determined from moments of the $f_i(\mathbf{x},t)$ population. In the absence of forces these are given by

$$\rho(\mathbf{x},t) = \sum_i f_i(\mathbf{x},t) = \sum_i f_i^{eq}(\mathbf{x},t), \tag{5}$$

and,

$$\rho(\mathbf{x},t)\mathbf{u} = \sum_i f_i(\mathbf{x},t)\mathbf{c}_i = \sum_i f_i^{eq}(\mathbf{x},t)\mathbf{c}_i, \tag{6}$$

respectively. Other relevant physical properties of pressure,

$$p(\mathbf{x},t) = C_s^2\rho(\mathbf{x},t), \tag{7}$$

and viscosity,

$$\eta = \rho C_s^2\left(\tau - \frac{\Delta t}{2}\right), \tag{8}$$

arise from the Chapmann-Enskog expansion process.

2.3. Boundary conditions

We encounter two different types of boundary conditions (BCs) within the LBM: a) the BCs at solid surfaces and b) the BCs at the inlets - outlets of the simulation domain. The former refers to enforcing the no-slip boundary condition on the velocity field, while the later on how to drive the fluid flow. Within the lattice Boltzmann framework these aim to establish certain conditions at a given boundary site by explicitly determining the values of the unknown post-collisional distribution functions propagating from unknown locations outside the geometry into the simulation domain after the streaming step, see eq. (3b).

2.3.1. *No-slip boundary conditions*

We enforce the no-slip boundary condition at solid surfaces (blood vessels' walls) by applying the mid-link bounce-back rule proposed by Ladd and Verberg [41]. Populations streaming towards solid nodes are reflected back towards the lattice nodes they came from, resulting in recovering the wall location (zero velocity) approximately half-way between the fluid and solid node.

The above choice combined with the single relaxation time collision model (BGK) can lead to viscosity dependent (relaxation time τ) location of solid surfaces [42]. For simulations involving blood flow within small-diameter blood vessels, e.g. arterioles and venules, a two-relaxation (TRT) or multi-relaxation (MRT) time collision model may be more suitable to ensure that blood flow is correctly resolved. Although this is a known issue for porous media flow, e.g. permeability calculations, which are sensitive to the exact location of solid surfaces [42], the sensitivity of blood flow simulations with respect to this aspect may be worth investigating as well. Furthermore, the TRT and MRT collision models can enhance the stability of LB simulations. For the purpose of this study, however, as well as for blood flow in large-diameter blood vessels, the BGK collision model suffices.

2.3.2. *Inlet - outlet boundary conditions*

This is probably one of the most challenging tasks when running blood flow simulations. While for regular geometries imposing inlet-outlet BCs is simple, for the case of vascular systems this information is difficult to obtain. Ideally, clinically measured flow rates should be imposed, which can be obtained non-invasively by using techniques such as Doppler Ultrasound or Magnetic Resonance Angiography [43]. However, this is not a straightforward task. Moreover, another challenge of simulations at full-human scale is the development of an efficient and accurate approach to imposing BCs on multiple outlets. Lo et al. [44] recently implemented a two-element Windkessel model in HemeLB_CPU to control the flow rate ratios at the outlets.

Within HemeLB_GPU, the blood flow is driven by applying either velocity or pressure BCs at the inlets and outlets, termed *iolets*, of the domain. BCs can be set independently for the two types of *iolets* encountered. Pressure BCs are imposed by assigning a target pressure at the boundary fluid nodes, which for the LBM is achieved through assigning the value of the local density (*ghost* density). For velocity BCs, we follow Ladd's method [45] to impose the expected velocity profile (e.g. parabolic, Womersley). This is based on a modification of the bounce-back scheme with a correction term (momentum exchange term) $-2w_i\rho\mathbf{u}\cdot\mathbf{c_i}/C_s^2$. A more detailed description on the implementation of BCs within HemeLB is provided by Nash et al. [46].

3. GPU code implementation

In this section we give an overview of the GPU code implementation. First we provide some general background information on the LB algorithm and the HemeLB code. Then we describe the steps taken to port the HemeLB code to GPU architecture.

3.1. *General background*

The LB algorithm proceeds in the following way:

1. Initialise macroscopic quantities, density ρ and velocity $\mathbf{u}$, required for the initialisation of the distribution functions f_i $(i = 0 - 18)$ to their equilibrium value f_i^{eq} using eq. (4).
2. Collision step: evaluate the updated distribution functions $f_i'(\mathbf{x}, t)$ according to eq. (3a).
3. Streaming step: the updated distribution functions $f_i'(\mathbf{x}, t)$ propagate to the neighbouring fluid site, $\mathbf{x} + \mathbf{c_i}\Delta t$, see eq. (3b).
4. Apply BCs: solid-fluid BCs and Inlet/Outlet BCs.
5. Repeat steps 2-4 timeSteps-times. These steps represent the core of the LBM algorithm.

The above scheme with the collision step preceding the streaming step is referred to as the Push-scheme [47]. It would also be possible to have the streaming step first, by gathering the distribution functions from the neighbouring fluid sites to a given fluid site and then perform the collision step. This is known as the Pull-scheme [29,47]. Further discussion on the above two schemes will follow in section 7.

The LBM is inherently amenable to parallelisation owing to its local nature and thus presents itself as a candidate for extreme parallelism on modern supercomputers. Exchange of data between neighbouring MPI ranks requires only nearest neighbour information and takes place during the streaming step, when the updated distribution functions stream in and out of the domain assigned to each MPI rank. The fluid sites of these shared edges are labelled as *domain-edge* sites, while the ones not requiring any exchange of information with neighbouring MPI ranks are labelled as *mid-domain* sites. Hence, a highly effective way of hiding the MPI communication overhead and enabling scaling of LBM algorithms up to extreme scales is by performing the following at each LBM iteration:

1. First, perform collision - streaming at *domain-edge* sites.
2. Then, issue the MPI exchange for the *domain-edge* sites.
3. Finally, perform collision - streaming at *mid-domain* sites, while overlapping these computations with the MPI data exchange.

HemeLB follows the same approach and registers the following steps to be executed at each iteration (timeStep) through the *StepManager* object

1. PreSend: Collision - streaming at *domain-edge* sites.
2. Send: Issue the MPI exchange of the updated distribution functions involved at *domain-edge* sites
3. PreReceive: Collision - streaming at *mid-domain* sites.
4. Receive: Wait for the MPI exchange of the updated distribution functions to complete.
5. PostReceive: Place the received distribution functions in the appropriate streaming location.
6. EndIteration: Perform necessary calculations at the end of the LB iteration for updating the property cache, containing macrovariables of interest, e.g. density, velocity etc. Finally, swap the distribution functions f^{old} (pre-collision) and f^{new} (post-collision). Essentially, only placing f^{new} into f^{old} is required.

3.2. *Porting the HemeLB code to GPU architecture*

Here we describe the steps taken to port the single component HemeLB_CPU code to GPU architecture. HemeLB_CPU is written in C++. The GPU version of HemeLB has been developed using the CUDA computing platform (CUDA C++) to run on NVIDIA's GPUs. We must note that, as we are moving into an era where there may be more kinds of GPUs available (NVIDIA, AMD, Intel), we would like to make the code eventually platform agnostic. To this direction, recent development efforts were carried out porting the CUDA code to the HIP runtime API, making it able to run on AMD GPUs as well. Here, however, we will restrict ourselves to the CUDA version of the code.

Given the complex nature of the existing CPU version of HemeLB, our initial attempt for porting the HemeLB code on GPU architectures focused on exporting the compute intensive parts onto the GPU (device), without making significant changes to the remaining structure of the code.

The steps taken involve the following:

1. Initialise the GPU.
2. Implement the GPU collision - streaming kernels.
3. Arrange the exchange of data between the device (GPU) and the host (CPU), i.e. the memory copies from the device to host (D2H) and from the host to device (H2D).
4. Arrange the CUDA streams for the various GPU operations and the appropriate synchronisation points (*cudaStreamSynchronize*).

3.2.1. *Initialisation of the GPU*

Initialising the GPU takes place at the beginning of the simulation and involves allocating memory and copying the data that will reside on the GPU for the duration of the simulation. More specifically this involves:

- Allocate memory on the GPU global memory and perform a H2D memory copy for the distribution functions (f^{old}, f^{new}) and the streaming indices; the later refer to the GPU memory locations that the post-collision populations will stream to. A change of the data layout is performed for these data structures. More details on this are provided in section 3.2.3.
- Allocate memory (GPU global memory) and perform a H2D memory copy for the following:
  1. information for boundary links' intersections (wall/iolet)
  2. information for iolets:
     (a) total number of iolets on local MPI rank
     (b) fluid sites' range associated with each unique iolet
     (c) normal vector at each iolet
  3. streaming indices for the received distribution functions f^{new} at shared edges
- Allocate memory (GPU global memory) related to BCs at iolets:
  1. for the case of Pressure BCs the *ghost* density for each iolet.
  2. for the case of Velocity BCs the correction term for each iolet fluid site.
- Copy to GPU constant memory the following:
  1. discrete velocity directions $\mathbf{c_i}$ of the LB model (D3Q19).
  2. inverse velocity directions (related to the bounce-back scheme).
  3. weights w_i for calculating f^{eq}.
  4. relaxation time τ and its inverse.

3.2.2. *Collision - streaming GPU kernels*

The compute intensive parts of the code that were exported on the GPU involve mainly the collision and streaming steps of the LB algorithm. These steps are performed together within the GPU kernels. This significantly reduces the data traffic to/from the GPU global memory, since there is only one read and one store of LB populations at every time step. Distribution functions and computations are in double precision. HemeLB distinguishes the following 6 types of collision – streaming, depending on the type of fluid sites and their corresponding streaming links: 1) Inner domain: only fluid sites without any links to any type of boundaries (walls/iolets), 2) Walls: fluid sites with a link to a solid surface, 3) Inlet, 4) Outlet, 5) Inlet with Walls and 6) Outlet with Walls. Hence, GPU CUDA kernels were initially implemented to account for each one for the above collision – streaming types. We must note that we have eventually merged the first two types of collision - streaming kernels, which provided a small gain in performance. All collision - streaming kernels can be potentially launched within the *PreSend* (*domain-edge* sites) and the *PreReceive* (*mid-domain* sites) steps of the code.

HemeLB groups the fluid sites in a consecutive ascending order depending on their collision - streaming type (indirect memory addressing [25,30,31]), with the corresponding fluid sites' range passed as an argument to the appropriate CUDA kernel. Depending on the LB stencil (D3Q19), each fluid site is connected to a set of neighbours. This connectivity information is computed at the beginning of the simulation and stored in the geometry input file (.gmy). When the CUDA kernels are launched, their threads (bundled into warps) are more likely to execute the same code branches, avoiding consequently any control divergence. Moreover, these kernels reside on different CUDA streams, see Fig. 1; hence, they can run concurrently and offer further acceleration of the computations. This is in contrast to the HemeLB_CPU code where the corresponding components execute in a serial manner.

We implemented the collision - streaming GPU CUDA kernels with the following assumptions:

1. Lattice model: D3Q19.
2. Collision operator: Single relaxation time approximation, i.e. BGK collision operator [39].
3. Wall boundary conditions: Mid-link bounce-back [41].
4. Inlet/Outlet boundary conditions: Any type of inlet and outlet BCs supported by HemeLB, i.e. pressure or velocity BCs, to drive the blood flow.

Each CUDA kernel performs the collision and streaming step, as well as applies the appropriate BCs when applicable, by determining the unknown populations f^{new} after the streaming step.

Each thread in the CUDA kernels performs the following:

1. Loads the f^{old} distribution functions from the GPU global memory into local memory.
2. Calculates the hydrodynamic macrovariables (ρ, $\mathbf{u}$), using eqs. (5), (6).
3. Calculates the equilibrium f^{eq} values (eq. (4)).
4. Calculates the post-collision distribution functions locally (eq. (3a)).
5. Performs the streaming step (eq. (3b)). If the thread encounters any boundary links during streaming (wall/iolet), apply the appropriate BCs to determine the unknown populations f^{new}.
6. Stores the updated values f^{new} into the GPU global memory.
7. Stores the hydrodynamic macrovariables (ρ, $\mathbf{u}$) into the GPU global memory at a specified frequency, e.g. every 100 time-steps, to decrease any unnecessary memory traffic to the GPU memory.

Populations that stream out of the simulation domain at domain edges during the streaming step are appended at the end of the f^{new} 1D array in *totalSharedFs*, see Figs. 2-3. The collision - streaming CUDA kernels for these sites are launched first during the step *PreSend*, as discussed in section 3.1. A CUDA synchronisation barrier (*cudaStreamSynchronize*) ensures that all GPU collision - streaming kernels in *PreSend* have completed their task. A D2H asynchronous memory copy (*cudaMemcpyAsync*) is then issued to transfer the populations in *totalSharedFs* to the host and arrange the MPI send to the appropriate neighbouring rank (step *Send*). Once transfered on the receiving MPI rank, a H2D asynch. mem. copy is issued and the data are placed in the *totalSharedFs* location of the f^{old} 1D array ((*PostReceive* step). Finally, a GPU kernel

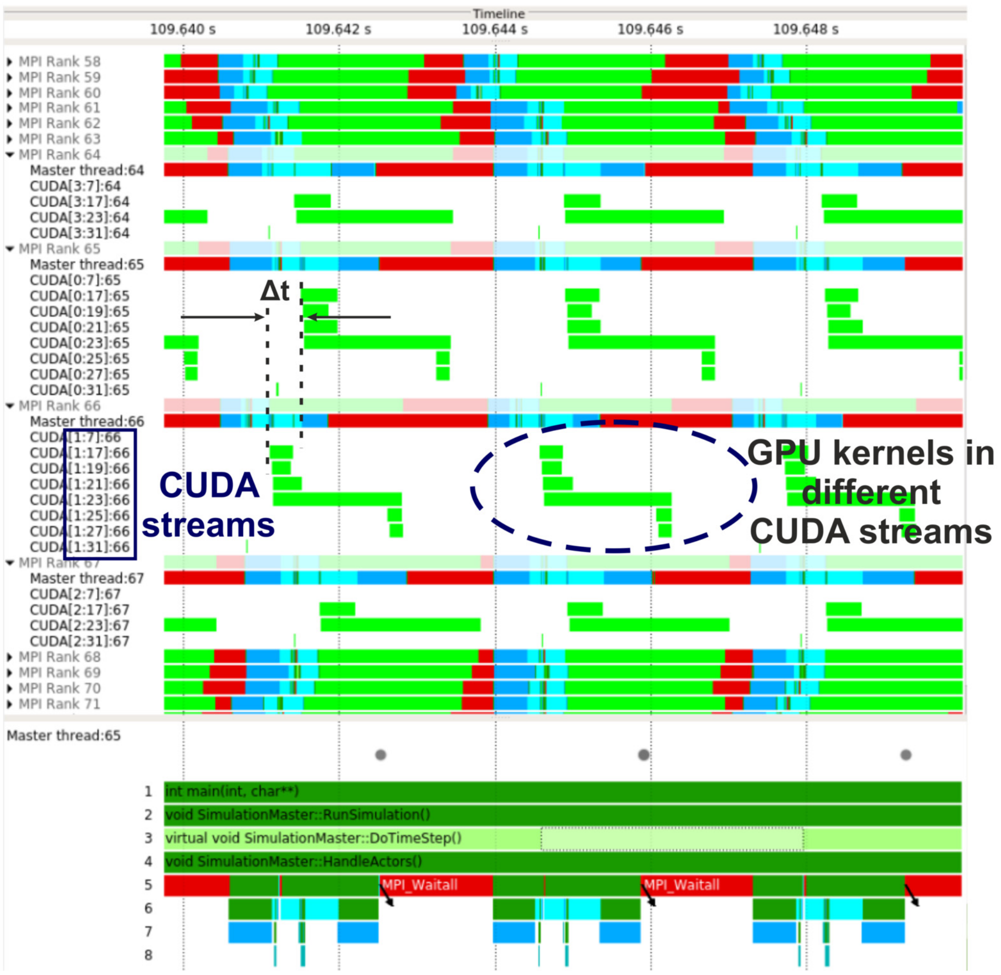


Fig. 1. HemeLB_GPU execution timeline: Focus of analysis zoom (3 timesteps) showing GPU CUDA kernels (green boxes) running concurrently in different CUDA streams for MPI ranks 64 to 67. Each line corresponds to a different CUDA stream. Some ranks/GPUs (MPI ranks 65 & 66) have additional kernels for processing iolets. This is a possible source of load imbalance. Here, ranks 64 & 67 have 2 kernels, compared to ranks 65 & 66 that have 6. Load imbalance leads to variations (Δt) in the starting time for the execution of the GPU kernels for varying ranks. Results from profiling a simulation on Juwels Cluster (Julich Supercomputing Centre) with 32 nodes and 129 MPI processes driving 128 Tesla V100 GPUs [48]. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

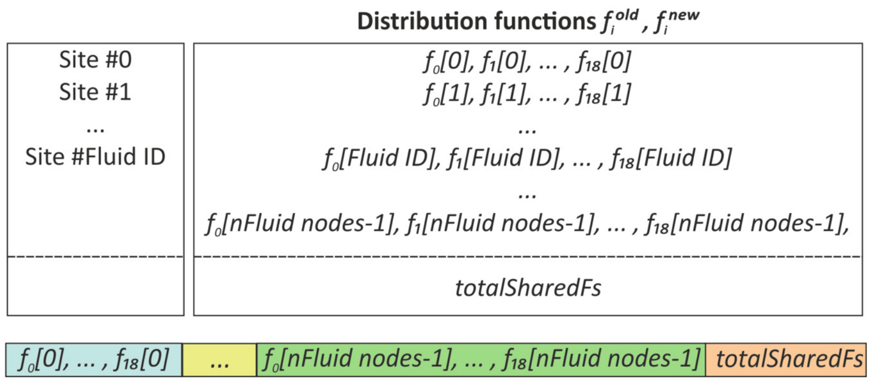

Distribution functions f_i^{old}, f_i^{new}
Site #0
Site #1
...
Site #Fluid ID
$f_0[0], f_1[0], \dots, f_{18}[0]$
$f_0[1], f_1[1], \dots, f_{18}[1]$
...
f_0[Fluid ID], f_1[Fluid ID], ... , f_{18}[Fluid ID]
...
f_0[nFluid nodes-1], f_1[nFluid nodes-1], ... , f_{18}[nFluid nodes-1],
totalSharedFs
$f_0[0], \dots, f_{18}[0]$
...
f_0[nFluid nodes-1], ... , f_{18}[nFluid nodes-1]
totalSharedFs

Fig. 2. *Array-of-Structures* scheme (HemeLB_CPU version). Distribution functions f^{old} (pre-collision) and f^{new} (post-collision) are arranged based on the fluid site ID in a 1D array.

(*GPU_StreamReceivedDistr*) performs the appropriate re-allocation of the received distribution functions, from the *totalSharedFs* in f^{old} into the destination buffer f^{new}.

Once the above re-allocation is complete and all collision - streaming GPU kernels in step *PreReceive* (*mid-domain* sites) have also completed their task (use *cudaStreamSynchronize*), swapping of the populations takes place in step *EndIteration*; this ends the LB algorithm for the current time-step.

3.2.3. Optimisation strategies

The main optimisation strategies used during the GPU code development were:

- **Change of data layout**. HemeLB stores the distribution functions (f^{old}, f^{new}) as an *Array-of-Structures* (AoS), where data is arranged in a 1D array in system memory based on the fluid site index, see Fig. 2. The GPU version stores these data fol-

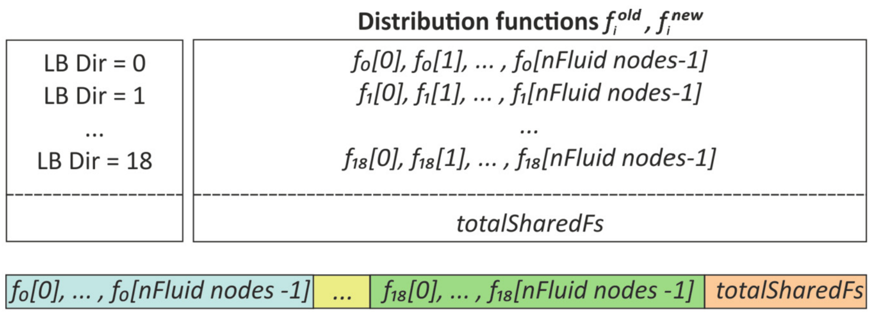

Distribution functions f_i^{old}, f_i^{new}
LB Dir = 0
LB Dir = 1
...
LB Dir = 18
$f_0[0], f_0[1], \dots, f_0[nFluid\ nodes-1]$
$f_1[0], f_1[1], \dots, f_1[nFluid\ nodes-1]$
...
$f_{18}[0], f_{18}[1], \dots, f_{18}[nFluid\ nodes-1]$
totalSharedFs
$f_0[0], \dots, f_0[nFluid\ nodes -1]$
...
$f_{18}[0], \dots, f_{18}[nFluid\ nodes -1]$
totalSharedFs

Fig. 3. *Structure-of-Arrays* scheme (HemeLB_GPU version). Distribution functions f^{old} (pre-collision) and f^{new} (post-collision) are arranged based on the discrete velocity directions of the lattice model (D3Q19) in a 1D array.

lowing the *Structure-of-Arrays* (SoA) scheme; data is arranged based on the LB discrete velocity directions, see Fig. 3. SoA scheme is more suitable for the GPU architecture as demonstrated by Tran et al. [29]. Finally the array appended at the end of the 1D arrays, labelled as *totalSharedFs*, that corresponds to the buffer for placing and receiving the post-collision distribution functions at shared edges remains the same as in the CPU version.

- **Change the sequence of steps** registered in *StepManager*, compared to HemeLB_CPU. HemeLB_GPU reorders the pattern of MPI exchanges of data. Instead of ordering the steps as *PreSend* → *Send* → *PreReceive*, the sequence is modified to *PreSend* → *PreReceive* → *Send*. Effectively, all GPU CUDA collision - streaming kernels (*PreSend*:*domain-edge* and *PreReceive*:*mid-domain*) are launched first, with control returned to the host, before issuing the MPI exchange of data. This allows a better overlap of the computations and improves consequently the code's performance.
- **Use of different CUDA streams for all GPU operations.** This allows overlapping the CUDA kernels' execution, see Fig. 1, and the asynchronous memory copies, Device to Host (D2H) and Host to Device (H2D), see Fig. 4.
- **Swap the distribution functions at the end of the LB iteration.** Exchange the pointers for the fundamental LB data (f^{old}, f^{new}), instead of explicitly swapping the data.

3.3. Compiling - running a simulation

Compiling the code is a two-stage process. First, the user must build the dependencies before compiling the source code. A detailed description of the input file and how to run a simulation is provided in the GitHub repository [49] and the official HemeLB website [50].

4. Large scale performance comparison

HemeLB has been specifically optimized to allow excellent strong scaling performance on the sparse and irregular geometries that are characteristic of vascular domains. In our comparisons of performance on CPU and GPU based architectures, we consider two particular domains of different resolutions. The first is a discretization of the full human venous tree consisting of approximately 1.6×10^9 lattice sites, whilst the second represents the circle of Willis arterial structure found in the brain possessing 10.2×10^9 sites. Whilst both of these domains are of significant magnitude, we demonstrate that even these are not sufficient to adequately occupy current petascale machines to their full extent. These domains are illustrated in Fig. 5.

In this study, we are focused on the strong scaling performance of HemeLB. Because of the fixed and irregular shape of vascular geometries we are considering, it is difficult to accurately partition the domain to ensure an even balance of work between processors necessary for a rigorous weak scaling analysis. Assessing and comparing strong scaling performance between CPUs and GPUs is not a straightforward task due to the inherent differences between their execution. Adding to this are the differences in how HPC facilities have packaged resources onto a node of their machines. In Table 1 we identify potential metrics for comparing the large-scale performance of CPU and GPU codes both in terms of scale (x-axis) and performance (y-axis) with their benefits and drawbacks.

A particular challenge in comparing the performance of GPU and CPU codes is the determination of an equivalence between the two architectures. By way of precedent, the Top500 [51] list of supercomputers gives a measure of total "cores" in a supercomputer. For machines accelerated by GPUs, it appears that the GPU subunit of a "streaming multiprocessor" (in NVIDIA nomenclature) is deemed to be equivalent to a CPU core. By this measure, a NVIDIA V100 GPU corresponds to 80 CPU cores, whilst an A100 GPU is equivalent to 108.

Through association with a number of research projects, HemeLB_CPU has been able to be executed on a range of HPC machines and architectures including:

- Blue Waters (NCSA - 22,636 nodes, 16 CPU cores on each node)
- ARCHER (EPCC - 4920 nodes, 24 CPU cores on each node)
- ARCHER2 (EPCC - 5848 nodes (Phase 1 = 1024 nodes), 128 CPU cores on each node)
- SuperMUC-NG (LRZ - 6480 nodes, 48 CPU cores on each node)

while HemeLB_GPU has been able to be executed on the following:

- Piz-Daint (CSCS - 5704 nodes, 12 CPU cores and 1 P100 GPU on each node),
- JUWELS-Cluster (JSC - 56 nodes, 40 CPU cores and 4 V100 GPUs on each node),
- JUWELS-Booster (JSC - 936 nodes, 48 CPU cores and 4 A100 GPUs on each node) and
- SUMMIT (ORNL – 4608 nodes, 42 CPU cores and 6 V100 GPUs on each node).

This breadth of machines has enabled us to develop a broad understanding of the HemeLB code and how it performs on a range of architectures. In our following analysis we will restrict reporting to cases where common test cases have been conducted.

4.1. HemeLB_CPU - strong scaling performance

As the capabilities of both HemeLB and supercomputers have increased, the strong scaling performance of HemeLB has been repeatedly demonstrated, from tens to hundreds of thousands of compute cores. Here we highlight this improvement on HPC facilities of varying architecture and scale. On the SuperMUC-NG

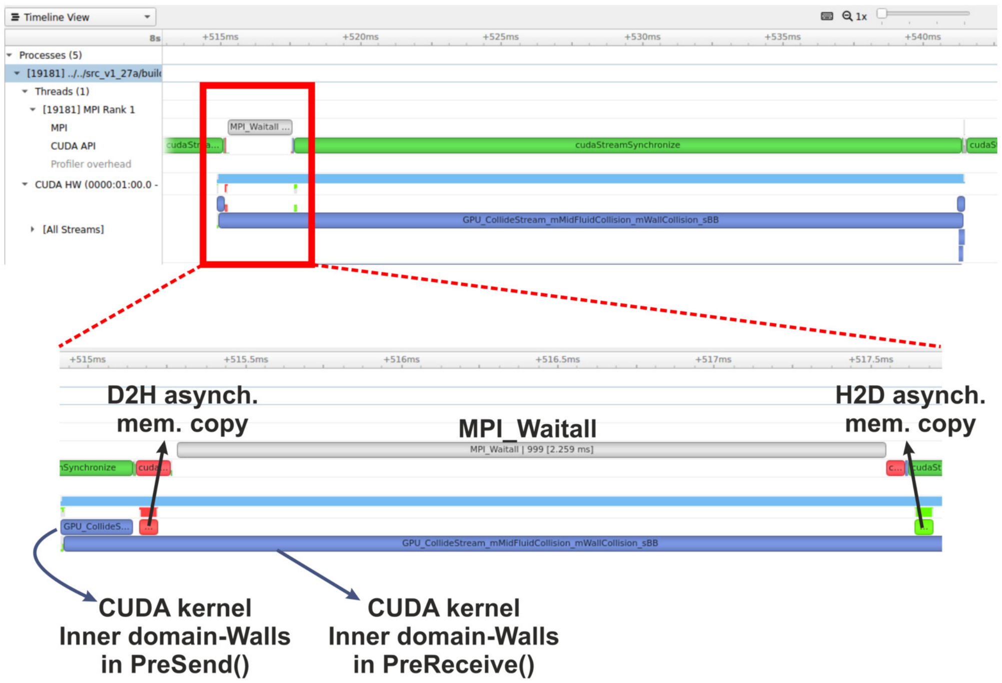


Fig. 4. Profiling the HemeLB_GPU code using NSight Systems. Focus of analysis zoom (1 timestep) showing GPU CUDA kernels (blue) in different streams, as well as the asynchronous memory copies (D2H and H2D). The D2H memory copy sends the data to the host to be sent to neighbouring MPI ranks; once the MPI send is complete and the updated distribution functions have been received on the host, the H2D mem. copy transfers these to the device to be placed in *totalSharedFs*. The MPI communications and the asynchronous memory copies (D2H and H2D) overlap with the computations of the *mid-domain* sites.

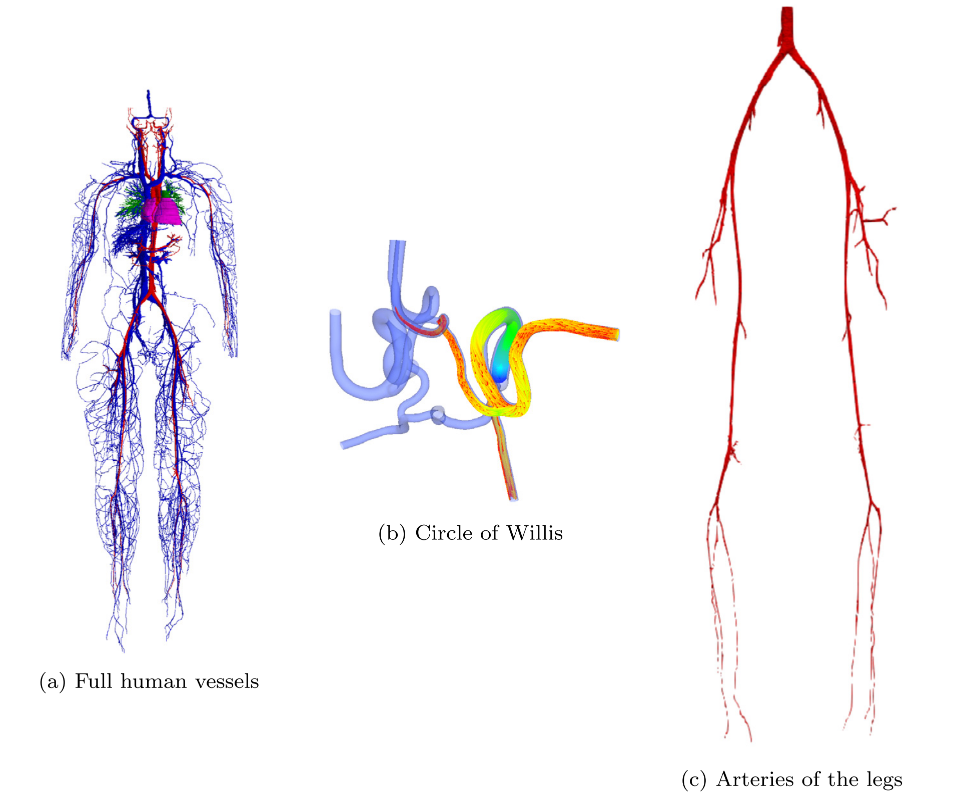


Fig. 5. Illustrations of the test domains used to test the large-scale performance of HemeLB's GPU and CPU versions: (a) The full human venous tree consisting of 1.6×10^9 lattice sites. (b) The circle of Willis (coW), arteries at the inferior side of the brain, consisting of 10.2×10^9 lattice sites. (c) The arterial legs consisting of 66.4×10^6 sites.

Table 1
Various potential metrics for assessing the strong scaling performance of CPU and GPU codes.

Metric	Type	Pros	Cons
Cores	Scale	Simple measure of number of cores deployed	Assumes equivalence between CPUs and GPUs; Reflects poorly on GPU codes; HPC facilities have far fewer GPUs than CPUs
Threads	Scale	Gives strong indication of total number of parallel processes utilised by a job	Assumes a GPU thread and a CPU core are equivalent; Reflects poorly on CPU codes
Nodes	Scale	Simple measure of total nodes used by a job	Resources available on a node varies widely between HPC facilities; GPUs may not be available on all nodes; Job may not demand all resources of a node
Wall Time	Performance	Easiest measure to record	Geometry dependent measure
Speed Up	Performance	Straightforward measure to interpret performance	Derived unit from wall time or MLUPS
MLUPS	Performance	More independent of geometry	Measure that is most relevant to LBM codes

machine in particular, we had the opportunity to test the performance of HemeLB on a new supercomputer that was then within the top 10 of the Top500 list and represented one of the closest estimates of performance on an exascale machine available. Independently and in collaboration with the POP Centre of Excellence [52], we were able to conduct a similar test regime on Blue Waters and SuperMUC-NG using the circle of Willis geometry - an arterial structure typical of production jobs. The details of these studies are more fully reported in [8,53,54]. In these studies, we were able to run simulations at up to near-full or full-machine scale on both machines (Blue Waters 288,000 cores, 80% of available cores; SuperMUC-NG 309,696 cores, 99.6% of available cores). We report the performance of these in Fig. 6. In Fig. 6, we also provide comparison to a similar test conducted on ARCHER with a smaller circle of Willis test domain of 777 million lattice sites. The improvement in performance through the use of a geometry that can better occupy the compute capacity of a machine is clear. We observed 75% strong scaling efficiency at 48,000 cores on ARCHER, 81% efficiency at 192,000 cores on Blue Waters and 75% efficiency on SuperMUC-NG at 147,456 cores. On all machines, HemeLB continues to scale strongly at all higher core counts tested. We believe that the tapering of performance at higher core counts can be attributed to two key reasons: 1) the test geometry not being large enough to fully occupy the compute cores at full machine scale; and 2) the impact of under-performing machine hardware or software components. These can impact the ability of machines to run large models effectively and can cause them to be unable to handle and display data generated from such simulations. Item (1) draws attention to an important issue which is only encountered by applications that seek to run models which require the full production partition in order to run: very often, the machines themselves may have not been optimised to routinely handle jobs that operate at this scale. This may mean that internode communication may take longer periods when large numbers of cores are being deployed. This makes it more difficult for computation to mask communication and good strong scaling performance to be achieved. Regarding item (2), on SuperMUC-NG, working with collaborators from POP CoE we identified several benchmark tests that exhibited significantly reduced performance due to a single faulty compute node. Even with such nodes excluded, the use of these extremely large core counts exposes benchmark tests to vagaries in performance that are frequently difficult and/or expensive to quantify.

4.2. HemeLB_GPU - strong scaling performance

We have been able to test the performance of the GPU code on Summit – the second fastest machine on the current Top500 list Top500 whose performance is accelerated by 27,648 NVIDIA V100 GPUs. Fig. 7 reports the strong scaling performance of HemeLB_GPU. The relative speed-up of simulations is evaluated with regards to the smaller amount of GPUs/CPUs that can accommodate the simulation domain being investigated in terms

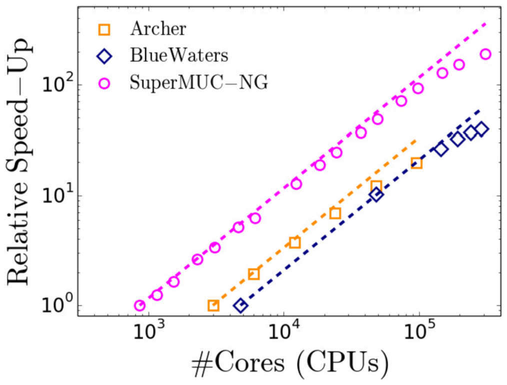


Fig. 6. Strong scaling performance of the CPU code on various generations of CPU based machines. Ideal scaling is indicated with the dashed lines.

of available memory. This is dictated by the GPU global memory. Using the circle of Willis geometry ($\sim 10.2 \times 10^9$ sites, see Fig. 5(b)) and comparing to a baseline measurement on 768 GPU cores, Fig. 7(a) demonstrates 90% perfect scaling performance on 6,144 GPUs and continued strong scaling to 18,432 GPUs, which is approximately 2/3 of Summit's capacity. The strong scaling efficiency drops to 72% at 12,288 GPUs and 60% at 18,432 GPUs. In an extra test on Summit, we examined the large-scale performance by increasing the simulation domain and examining the scaling characteristics of a cylinder constructed of $\sim 37.5 \times 10^9$ sites, labelled as ExaPipe geometry (baseline 3,072 GPUs). As shown in Fig. 7(a), this improves the strong scaling performance and leads to 74% strong scaling efficiency at 18,432 GPUs, due to the increased computation to communication ratio. This makes evident the improvement in performance through the use of a geometry that can better occupy the compute capacity of the machine.

Investigating further the deviation from perfect strong scaling (solid line in Fig. 7(a)), we examined the time per LB iteration, as this is indicative as to whether computations can mask the MPI communications. Results with the circle of Willis geometry ($\sim 10.2 \times 10^9$ sites, Fig. 5(b)) reveal a drop below 75% strong scaling when the time per iteration $t_{LB}^{iter} \sim 3.8 \times 10^{-3}$ s and 12,288 MPI tasks. With the ExaPipe geometry ($\sim 37.5 \times 10^9$ sites) this drop takes place at $t_{LB}^{iter} \sim 8.7 \times 10^{-3}$ s and 18,432 MPI tasks. Generally, deviations from excellent strong scaling will start to emerge when the computational time at *mid-domain* sites becomes comparable in size and cannot mask either of the following: (i) the communication overhead, which typically increases with the number of processes used; (ii) the variations in the starting time of each LB iteration due to either a) load imbalance issues exposed at large core counts, as the number of fluid sites per GPU decreases (at the time-scale of 10^{-3} s for example from Fig. 1) or b) hardware issues. The MPI latency on Summit, as reported in [55], is of the order of 10^{-6} s to 10^{-5} s; hence, the most probable cause of the deviation from strong scaling is load imbalance. This was identified

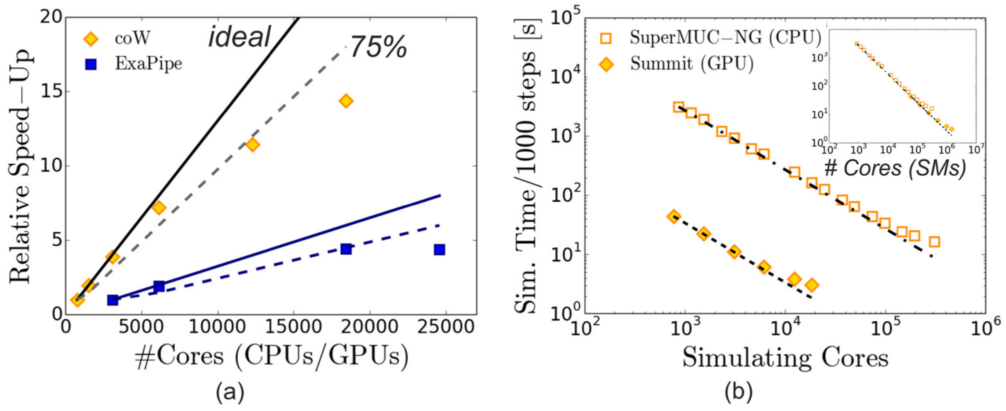


Fig. 7. Strong scaling performance of HemeLB: (a) Relative speed-up of HemeLB_GPU on Summit. The baseline measurement, against which the relative speed-up is evaluated, corresponds to 768 CPUs/GPUs for the circle of Willis (coW) and 3,072 CPUs/GPUs for the ExaPipe geometry. (b) Comparison of strong scaling performance of HemeLB_CPU and HemeLB_GPU codes using a common geometry (circle of Willis). The normalised walltime recorded by each architecture is plotted against the raw number of resources (CPU/GPU cores) requested. A speed-up factor of approximately 85 is observed. Inset: Normalised walltime as a function of CPU equivalent hardware units (CPU cores and SMs).

to account for the deviation from ideal strong scaling when profiling a simulation with the legs' arteries (see Fig. 5(c)) on Juwels Cluster (JSC) with 32 nodes and 129 MPI processes driving 128 V100 GPUs [48]. This is demonstrated in Fig. 8, which reveals considerable variations for the total kernels' execution time for varying MPI ranks/GPUs. Mid-range ranks are taking much longer, especially for collision-streaming at the *domain-edge* sites. Furthermore, the most heavily overloaded ranks have increased execution times due to processing of iolets at *domain-edge* sites. Here, we have used a basic decomposition of all geometries tested, which however returns an unbalanced work-load between MPI ranks. Specific load-balancing libraries like Zoltan [56] and ParMETIS [57] could help in this direction. However, the extensive pre-processing when using these libraries leads to substantially longer walltimes and more memory, making these impractical to rely on.

To enable comparison of the HemeLB_CPU and HemeLB_GPU codes we utilised the same circle of Willis geometry as on Blue Waters and SuperMUC-NG. Fig. 7(b) provides a side-by-side comparison of the SuperMUC-NG and Summit results using an equivalence metric of 1 CPU core to 1 GPU. Here we can observe that the acceleration of the GPU code for an equivalent number of "cores" leads to a speed-up factor of approximately 85. For both codes, the tapering of results at the highest core counts is reflective of the limitation of the size of the geometry at these scales, as discussed above.

Due to the fundamental nature of the GPU architectures, a measure of cores is not necessarily the most relevant measure of parallelism in such applications. In the inset of Fig. 7(b) we compare the performance of the CPU and GPU codes using the equivalence suggested by the Top500 list, where NVIDIA V100 GPUs are equivalent to 80 CPU cores. Probably this arises from the fact that each Tesla V100 features 80 streaming multiprocessors (SMs). Here, it can be seen how the GPU code is able to continue the scalability characteristics to a higher level of parallelism than is achieved with the CPU codes. The total level of parallelism exhibited by the GPU code can be estimated by taking into account the fact that the NVIDIA V100 GPUs available on Summit each possess 5120 FP64 CUDA cores (double precision) that execute tasks in parallel. As such, the performance we report on Summit represents 90% efficient scaling on over 31 million parallel tasks and continued strong scaling behaviour up to over 94 million processes. This helps to support our argument that the inherent parallelism of HemeLB is well positioned to be deployed on an exascale machine.

In conjunction with the scaling tests discussed above, we have also generated some more normalised performance data for the

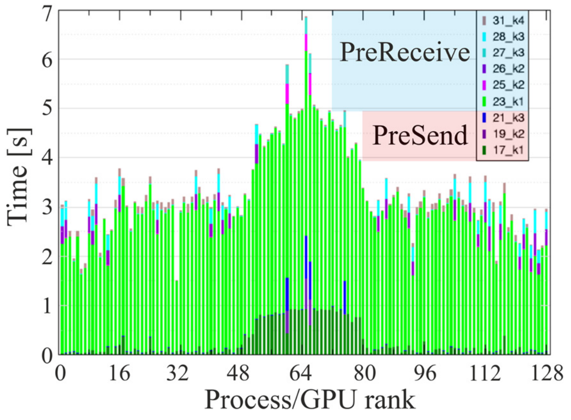


Fig. 8. Total GPU kernels' execution time for varying MPI ranks/GPUs from profiling a simulation with the legs' arteries (see Fig. 5(c)) on Juwels Cluster (JSC) with 32 nodes and 129 MPI processes driving 128 V100 GPUs [48]. Considerable variations are observed by MPI rank, with mid-range ranks taking much longer especially for collision-streaming at the *domain-edge* sites (*PreSend* step). The GPU kernels are labelled as follows: 1) k1: collision-streaming at *Inner domain & Walls*, 2) k2: collision-streaming at *iolets*, 3) k3: collision-streaming at *iolets* with *walls* and 4) k4: kernel for streaming the received distribution functions from neighbouring MPI ranks. The numbering in front of the kernel type corresponds to the CUDA streams. The most heavily overloaded ranks have increased computational times due to processing *iolets* (kernels k2 & k3) in *PreSend*.

CPU and GPU versions of HemeLB on a wider variety of machines. Fig. 9 reports the performance of simulations in Millions of Lattice site Updates Per Second (MLUPS) on a per core and node basis. MLUPS is defined as

$$\text{MLUPS} = \frac{nFluidSites \times nTimeSteps}{SimTime \times 10^6} \tag{9}$$

where $nFluidSites$, $nTimeSteps$ and $SimTime$ are the total number of fluid sites in the simulation domain, number of time-steps and total simulation time in seconds respectively. The evaluation of the performance on a per core (MLUPSpc) and per node (MLUPSpn) basis is based on the following definitions

$$\text{MLUPSpc} = \frac{\text{MLUPS}}{nCores} \quad \text{and} \quad \text{MLUPSpn} = \frac{\text{MLUPS}}{nNodes} \tag{10}$$

where $nCores$, $nNodes$ are the total number of CPU cores (MPI ranks) and total compute nodes respectively. Here, we must note that we used one MPI rank per GPU, although HemeLB_GPU can also run using other configurations as well by having multiple MPI ranks accessing the same GPU. Regarding the simulations reported

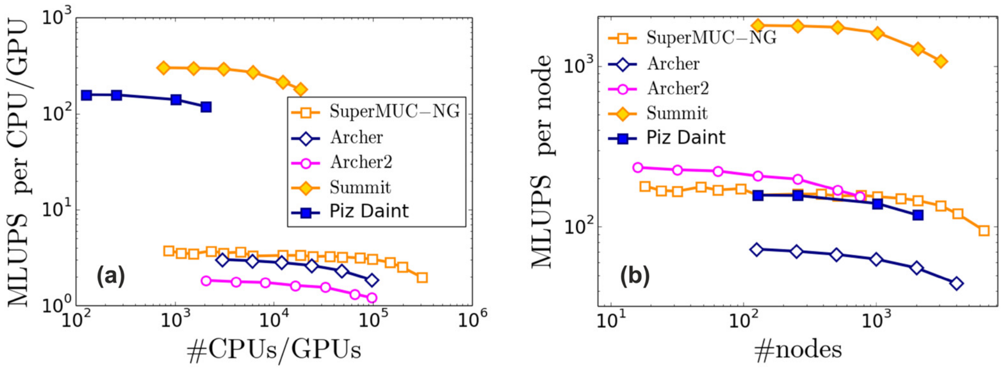


Fig. 9. Performance of HemeLB reported in millions of lattice sites per second (MLUPS) on (a) a per core basis, MLUPSpc, and (b) per node basis MLUPSpn. Open symbols correspond to results from CPU-only HPC machines (HemeLB_CPU), while the ones with filled symbols from HPC machines with NVIDIA GPUs (HemeLB_GPU). Simulations were conducted using a range of domains: small circle of Willis - ARCHER; large circle of Willis - Blue Waters, SuperMUC-NG, Summit; Full human veins - ARCHER, ARCHER2, Piz Daint.

in Fig. 7(a) on Summit, the baseline measurement of HemeLB_GPU with: a) the circle of Willis geometry is ~ 300 MLUPSpc (a total of 2.302×10^5 MLUPS on 768 CPUs/GPUs) and b) the ExaPipe geometry is ~ 316 MLUPSpc (a total of 9.703×10^5 MLUPS on 3072 CPUs/GPUs). For comparison reasons, the performance of HemeLB_CPU on SuperMUC-NG with the circle of Willis is ~ 3.5 MLUPSpc.

4.2.1. *HPC configurations - impact on code performance*

The above is informative as we have access to a broad range of high-end machines with different configurations of CPUs, GPUs, memory and bandwidth. For all machines, the drop in performance at higher node counts is due to the test domain not being sufficiently large to occupy the resources at that scale, the point at which this occurs varying between machines due to the different memory availability on CPU cores and GPU cards. Here we can see how the construction of a node can impact performance. For example, Piz Daint, a machine with only a single NVIDIA P100 GPU per node, delivers comparable performance to a single, CPU only, node on SuperMUC-NG. By comparison, the nodes on Summit, containing 6 NVIDIA V100s, deliver a performance improvement of a factor of at least 10 on other reported machines – i.e. 1 Summit node delivers the same performance as 10 SuperMUC-NG nodes. This indicates that the largest data for Summit reported in Fig. 7 corresponds to over 30,000 SuperMUC-NG nodes – 5 times larger than the actual SuperMUC-NG.

The comparison between the 2013-era ARCHER machine and the current generation ARCHER2 provides another notable comparison. Here, with more than 5 times as many cores per node, the new machine can only generate a node-based performance improvement of a factor of 2.9. This is due to hardware decisions that were made during procurement that led to a net reduction in memory bandwidth per core. Such decisions can have a negative impact on the relative performance of monolithic codes, especially those that are memory bound such as the lattice Boltzmann method. However, the same decisions can also be advantageous for other compute patterns that are less dependent on such hardware choices. The original ARCHER machine possessed 2×12-core Intel Ivy Bridge E5-2697 v2 CPUs on each node whilst the newer ARCHER2 has 2×64-core AMD Zen2 7742 CPUs per node. Whilst the AMD CPUs do run at a lower frequency than the Intel cores, this is not sufficient to fully explain the per-core difference in performance between the two machines. On earlier generations of Intel and AMD hardware, a similar difference in performance for a lattice Boltzmann code was reported in [58].

The variation in performance on a per core and node basis reported in Fig. 9 highlights the challenges posed when designing HPC architectures. Choices such as the number of GPUs and CPUs

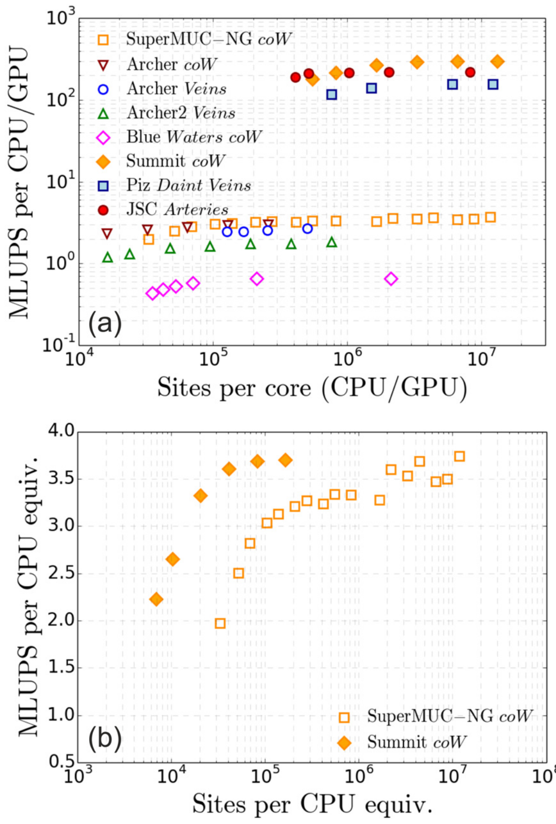


Fig. 10. Performance of HemeLB in MLUPS versus number of fluid sites: (a) per hardware unit (CPU/GPU), i.e. MLUPSpc and (b) per CPU core equivalent, i.e. scaling both performance (MLUPS) and fluid sites using the equivalence suggested by the Top500 list, which depends on the number of Streaming Multiprocessors (SMs) a GPU consists of. *coW*, *Veins* and *Arteries* refer to the domains illustrated in Fig. 5. One challenge of achieving effective exascale simulation will be to generate geometries that are large enough to effectively occupy the hardware. A drop in compute performance is observed at various values of sites per core on both CPU and GPU architectures. The point at which this occurs varies between machines.

deployed on a node, memory availability and network will all have an impact on code performance and will vary depending on different codes' operational requirements. Equally, different compute patterns will also pose competing demands on HPC performance.

5. Challenges at the emerging exascale

Based on our current investigations, the compute and thus scaling performance of HemeLB is dependent on the number of lattice

Table 2
Minimum memory demands for large-scale geometry generation in HemeLB on a CPU only machine, all figures in TB.

Cores	Classification	10^5 sites/core	$5\text{x}10^5$ sites/core	10^6 sites/core
10^5	Current HPC - medium scale	12.72	63.6	127.2
$5\text{x}10^5$	Current HPC - largest scale	63.6	318	636
10^6	Next-Generation HPC	127.2	636	1272
10^7	Exascale HPC	1272	6360	12720

sites per core. This is proportional to the computational time per LB iteration; in the case that this is sufficient to hide the MPI communication, then the strong scaling behaviour can be maintained. This is demonstrated in Fig. 10, where we plot the simulation speed against the number of sites per core for the CPU code. A significant performance drop is seen below $\sim 10^5$ sites/core for HemeLB_CPU and $\sim 10^6$ sites/core for HemeLB_GPU, demonstrating this to be a limit at which computation is outweighed by communication between nodes.

We believe that a full human vascular model may require at least 50×10^9 lattice sites to resolve the model at high resolution. Such a geometry, being five times larger than that used in the POP evaluation [53], will enable many of the performance concerns seen at scale on SuperMUC-NG to be addressed. The challenges associated with a) creating, b) storing and c) simulating a geometry of this scale are highly non-trivial. Regardless of the processing method chosen, storage requirements can be on the order of tens of terabytes for interim files (see Table 2 for the minimum memory demands for large-scale geometry generation in HemeLB on a CPU only machine). Furthermore, processing time may exceed 50 hours; both (storage requirements and processing times) are factors that may exceed allocation fair-use limitations imposed by HPC facilities. For future exascale machines with some 10 million cores, the potential requirement of 10^{12} lattice sites to achieve good scaling will further challenge the mesh generation of HemeLB (and indeed any code) and the machine's storage infrastructure. The HemeLB developers remain in collaboration with operators of SuperMUC-NG to develop solutions for addressing these concerns.

Whilst the figures presented here are particular to our deployments of HemeLB, they remain illustrative of the challenges of migrating monolithic codes to exascale HPC facilities. In addition to the storage challenges related to generating geometries of sufficient scale, the operational demands of loading such a domain for simulation, writing and storing output will put further demands on the resources of an exascale machine. Furthermore, the time required for pre- and post-processing operations is an additional challenge for operating efficiently at scale.

The scaling data presented here focuses on the simulation phase of the code. Demands of data writing and post-processing in particular may be harder to predict in general, as the output requirements can be specific to a particular problem. The tools used for pre- and post-processing may not have received the same degree of performance optimization as the simulation kernels; this highlights the need to consider the whole simulation workflow when evaluating the performance of a code. The use of workflow management tools such as RADICAL CyberTools [59] can help to manage multiple processes concurrently by reducing resource wastage; this is particularly true for ensemble type simulation studies. For the post-processing of HemeLB data an ongoing collaboration with the LRZ Centre for Virtual Reality and Visualisation and Intel has developed a workflow based on Intel OSPRay Studio to enable rapid visualisation of the very large and complex datasets that we can routinely generate [60]. Running this on HPC systems as part of a computational workflow enables an immersive rendering of vascular data to be obtained. Future work in this direction envisions the development of a 3D virtual reality experience of simulation data and the capacity for computational steering of a running simulation. As the output data structures generated by the CPU and GPU codes are the same, this tool is compatible with both versions of the code. An example of the results that can be achieved with this workflow is demonstrated for the circle of Willis case study in Fig. 11.

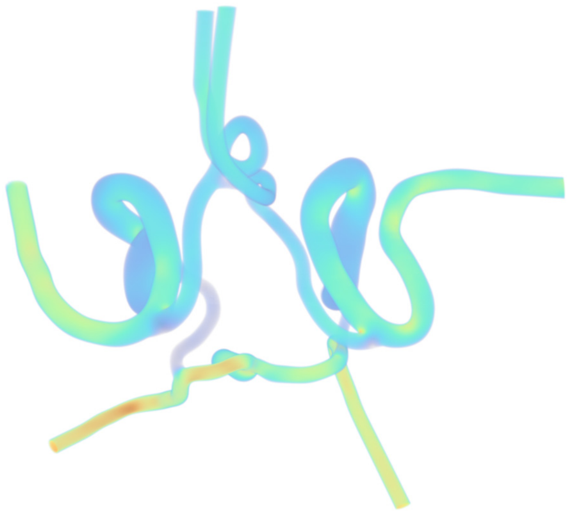


Fig. 11. Example visualisation of a velocity field within the circle of Willis domain using the workflow developed in a collaboration with LRZ and Intel. This demonstrates how high performance computers can be deployed to efficiently render images of the very large data sets created by HemeLB with human-scale data. As the data structures output by the CPU and GPU versions of the code is the same this tool can be used with both.

The ability for a given application to achieve optimum performance on a particular machine will require an ongoing co-design effort between both code developers, HPC operators and hardware experts. As the specific choice of hardware and its deployment can have significant impacts on the performance of a code, it is incumbent upon operators to ensure that the choices that they make during procurement bring the greatest benefit to the widest cross-section of their users of all application types. There is an equal onus on code developers to be looking forward to optimisations and performance gains that can be obtained on a given piece of hardware such as through compiler options as new options become available, or through an update of the code itself. These demands must also be balanced against the need to generate scientific knowledge from a code, where the ability to scale efficiently to large resource counts may not be the most decisive criterion.

6. Blood flow simulations with the HemeLB_GPU code

Here we provide a simple example to demonstrate the HemeLB_GPU code's capacity of running blood flow simulations by examining flow through a simple cylindrical pipe. Such a domain provides a canonical representation of the majority of blood vessels that we study with HemeLB and allows for a straightforward comparison between output generated by the CPU and GPU implementations.

The geometry input file format of the GPU version of the code is identical to that used for the CPU version. This requires voxeli-

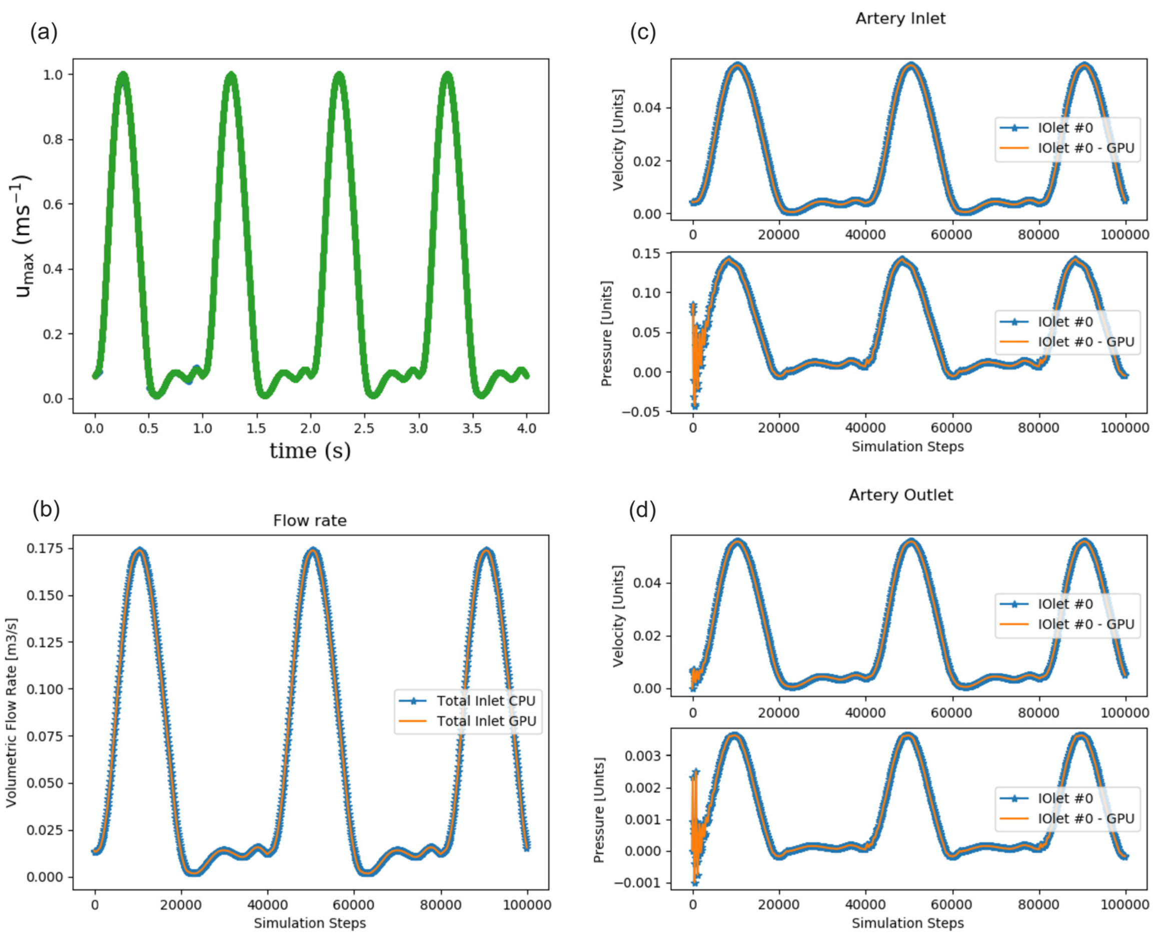


Fig. 12. Comparison of HemeLB_CPU and HemeLB_GPU codes. Simulation results using a pipe domain with velocity BCs imposed at the inlet and fixed constant pressure at the outlet. (a) The maximum velocity profile imposed at the inlet, corresponding to a heart beat profile of 60 beats per minute. (b) Comparison of the volumetric flow rates. (c) Comparison of maximum velocity and pressure at the inlet. (d) Comparison of maximum velocity and pressure at the outlet.

sation of a .stl file, representing the domain, to a .gmy file format, which stores the relationships between each site on the lattice and its neighbouring directions (indirect addressing). A suitable .stl file will represent the walls of the simulation domain, as well as the iolets by 2D planar openings, with their corresponding normal vectors oriented inwards towards the fluid. This conversion is undertaken using a set of scripts available in the open-source HemePure_Tools repository [61] and is built off features from the open-source Palabos code [11].

The simulation reported in Fig. 12 is driven by a heartbeat velocity profile of 60 beats per minute at the inlet (see Fig. 12(a)). This is implemented following Ladd [45]. The profile shown indicates the maximum velocity within the inlet; this is scaled by a weighting factor at each individual inlet site. In this case a parabolic profile was used to determine this weighting value and varied from 1.0 at the centre to 0.0 at the walls. Our simulation domain possessed a radius of 1 mm and a length of 10 mm and was discretized with a lattice grid spacing of dx = 0.1 mm. The simulation was run for 100,000 timesteps, with a timestep of dt = 2.5×10^{-5} sec; the simulation had a relaxation time of $\tau = 0.53$. A fixed constant pressure, implemented using the Nash method [46], is imposed at the outlet boundary. We compare the simulation results produced by HemeLB_CPU and HemeLB_GPU. The output format of both implementations is returned in an identical compressed format. This can be converted to a human readable format using the open-source *hemeXtract* tool [62]. Fig. 12(b)-(d) demonstrate that the output obtained from the two versions of the code is identical.

7. Future work

We outlined here a first implementation of a GPU accelerated version of HemeLB using CUDA C++. Results demonstrate that the code continues to exhibit excellent strong scaling performance up to thousands of GPUs and a decent speed up compared to HemeLB_CPU. This is an excellent outcome; however, code development and performance optimisation is an ongoing process, considering that single phase LB codes have demonstrated an even further gain in performance in terms of MLUPS per core.

Our initial efforts focused on exporting the collision-streaming steps of HemeLB on the GPU, without making significant changes to the rest of the code. Furthermore, trying to take advantage of how GPUs read from global memory, we changed the data layout for the distribution functions, f^{old} and f^{new}, from the AoS scheme to SoA. This allows for contiguous memory access within a warp during reading of the populations at the beginning of the collision-streaming GPU kernels. The same though does not apply during writing of the populations for the streaming step. The cost of this uncoalesced memory access can be reduced by considering a more appropriate data storage scheme. Herschlag et al. [30], for example, found strong evidence that semi-direct addressing is superior for arterial and porous media geometries using LB simulations, with a performance boost when implementing a Collected Structure of Arrays (CSoA) [63–65] data layout scheme. CSoA holds smaller sub-collections of the distribution functions with a stride length s and can help achieve more aligned writes on the GPU during the streaming step. This data layout significantly outperformed the SoA scheme, almost by a factor of two, when tested on the Summit supercomputer (ORNL - V100 GPUs) [30]. CSoA reduces to the AoS (SoA) scheme when choosing a stride length of $s = 1$ (number of the fluid sites).

Future work may also focus on various other issues, including the following:

1. Optimise the GPU kernels' performance. Tools and metrics provided by NVIDIA Nsight Compute [66] and Nsight Systems [67] can help to this end. Recent optimisations at the main GPU collision-streaming kernel level (*Inner domain & Walls*), through our participation in the UK National GPU Hackathon 2022 event, led to a performance gain of $\sim 50\%$ when tested on NVIDIA A100 GPUs on Baskerville (at the University of Birmingham). The optimisations applied involved: a) loop unrolling (`#pragma unroll 19`), b) merging of loops and c) surprisingly, increasing the maximum number of registers per thread (compiling with the option `-maxrregcount 200`). Generally, the available registers per streaming multiprocessor (SM) impose a restriction on the number of active warps on the SM and consequently may impact the GPU occupancy, leading to performance degradation. Here, however, despite decreasing the GPU occupancy by increasing the registers per thread (87 up from 40) led to a performance boost as this enabled access of the threads to local memory, decreased the stalled warps' cycles and reduced the memory traffic to the GPU global memory, which is much slower. Results from profiling the kernel using NVIDIA Nsight Compute [66] are shown in Fig. 13. The enhancements in kernel performance are evident, with an increase in the Streaming Multiprocessor (SM) compute throughput and a decrease in the Memory throughput. The roofline analysis, shown in Fig. 13(b), reveals an increase in the double precision arithmetic intensity. Further work is currently underway to optimise all the GPU kernels.
2. Convert from the Push to the Pull-scheme [29,47]. This refers to modifying the sequence of the streaming and collision steps of the LB algorithm. The Push-scheme, currently implemented, refers to the situation when the collision precedes the streaming step, while the reversed situation is known as the Pull-scheme. The fundamental difference of the two schemes lies with the ordering of uncoalesced memory accesses during reading from or writing to the GPU global memory. It was shown that the Pull scheme performs better than the Push-scheme, due to the cost of uncoalesced memory accesses during reading from the GPU device memory (Pull-scheme) being lower than during writing (Push-scheme) [29,30,47,68,69]. For example, Herschlag et al. [30] report speed ratios between pushing and pulling over a range of kernels and GPUs, with a ~25 to 35% speed up for indirect addressing and SoA on the P100 and V100 GPUs respectively.
3. Improve boundary conditions at iolets. Currently HemeLB_GPU supports driving the fluid flow by using either pressure or velocity boundary conditions, which can vary as a function of time. More physiologically correct boundary conditions should be applied though, when running patient- specific flow simulations, especially for outlet BCs to account for downstream resistance [32,70,71]. However, these resistance models require feedback from experiments and iterative tuning at each outlet, so that eventually the simulated and *in vivo* volumetric flow rates can match.
4. Implement elastic walls to better represent realistic vascular geometries, following our recent implementation in HemeLB_CPU, which most importantly demonstrated that this is feasible without a loss of computational performance [72].
5. Add the option two-relaxation (TRT) and multi-relaxation (MRT) collision operators.

As exascale HPC becomes available, these improvements, in conjunction with the capability described in this paper, will continue to ensure that HemeLB is able to take full advantage of these machines for deployment in human-scale blood flow simulations. In an effort to reach this goal for use in the field of personalised medicine, we have developed a self-coupled version of HemeLB that allows simultaneous study of arterial and venous domains [8] and provided an illustration of how it can be used in problems of clinical interest [73]. The enhanced performance capabilities of the GPU code will enable these CPU implementations to be conducted more quickly whilst retaining HemeLB's known strong scaling characteristics.

8. Conclusions

In this paper we have outlined a version of HemeLB capable of execution on NVIDIA GPUs that delivers excellent strong scaling performance to tens of thousands of GPUs. This has been built off a CPU-only version of HemeLB that has demonstrated similar strong scaling performance to the full production partition of the German supercomputer SuperMUC-NG. We have examined the new code's performance on a number of difference HPC platforms that represent a broad spectrum of hardware manufacturers and deployment frameworks. We illustrate that the performance of a code can be highly dependent on these factors with newer configurations not necessarily yielding better performance.

As the arrival of exascale machines moves closer, we will continue to develop all versions of HemeLB to aim for best performance on the widest range of machines. For example, the CPU code will look to incorporate the latest optimizations for code compilation and performance. Similarly we will continue to optimize the GPU port and aim to convert from our current CUDA-based implementation to one that allows deployment on both NVIDIA and AMD GPUs. This second objective will allow us to run HemeLB on a wider range of GPU accelerated HPC. As new features and libraries become available, we anticipate that further optimizations can be made in the GPU code to further reduce communication time at scale and further improve the scaling characteristics of our tool.

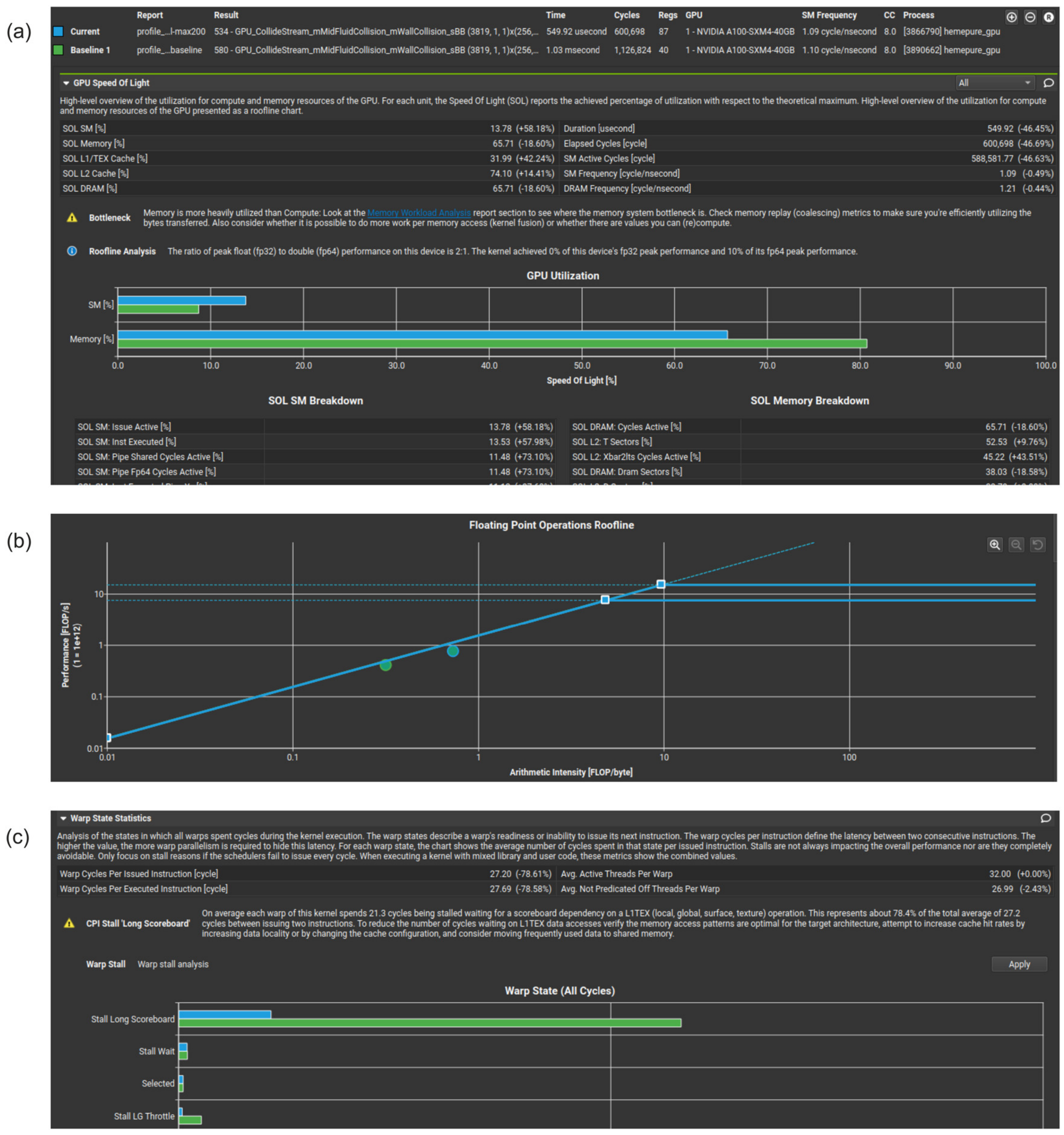
The results that we have presented in this paper highlight the role that all stakeholders have in determining the specifications of future HPC facilities. This can be regarded as a co-design exercise that will be beneficial to both code developers and HPC operators alike. Achieving a balance between the hardware demands of different compute patterns will be critical to generating the best outcomes for a wide range of scientific applications. Making future users of a new HPC facility aware of the hardware design as early as possible will let code developers adapt to any changes that may be necessary.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.



	Report	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
🟦 Current	profile_...l-max200	534 - GPU_CollideStream_mMidFluidCollision_mWallCollision_sBB (3819, 1, 1)x(256,...	549.92 usecond	600,698	87	1 - NVIDIA A100-SXM4-40GB	1.09 cycle/nsecond	8.0	[3866790] hemepure_gpu
🟩 Baseline 1	profile_...baseline	580 - GPU_CollideStream_mMidFluidCollision_mWallCollision_sBB (3819, 1, 1)x(256,...	1.03 msecond	1,126,824	40	1 - NVIDIA A100-SXM4-40GB	1.10 cycle/nsecond	8.0	[3890662] hemepure_gpu

GPU Speed Of Light

All

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

SOL SM [%]	13.78 (+58.18%)	Duration [usecond]	549.92 (-46.45%)
SOL Memory [%]	65.71 (-18.60%)	Elapsed Cycles [cycle]	600,698 (-46.69%)
SOL L1/TEX Cache [%]	31.99 (+42.24%)	SM Active Cycles [cycle]	588,581.77 (-46.63%)
SOL L2 Cache [%]	74.10 (+14.41%)	SM Frequency [cycle/nsecond]	1.09 (-0.49%)
SOL DRAM [%]	65.71 (-18.60%)	DRAM Frequency [cycle/nsecond]	1.21 (-0.44%)

Bottleneck Memory is more heavily utilized than Compute: Look at the Memory Workload Analysis report section to see where the memory system bottleneck is. Check memory replay (coalescing) metrics to make sure you're efficiently utilizing the bytes transferred. Also consider whether it is possible to do more work per memory access (kernel fusion) or whether there are values you can (re)compute.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% of this device's fp32 peak performance and 10% of its fp64 peak performance.

SOL SM Breakdown

SOL SM: Issue Active [%]	13.78 (+58.18%)
SOL SM: Inst Executed [%]	13.53 (+57.98%)
SOL SM: Pipe Shared Cycles Active [%]	11.48 (+73.10%)
SOL SM: Pipe Fp64 Cycles Active [%]	11.48 (+73.10%)

SOL Memory Breakdown

SOL DRAM: Cycles Active [%]	65.71 (-18.60%)
SOL L2: T Sectors [%]	52.53 (+9.76%)
SOL L2: Xbar2lts Cycles Active [%]	45.22 (+43.51%)
SOL DRAM: Dram Sectors [%]	38.03 (-18.58%)

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	27.20 (-78.61%)	Avg. Active Threads Per Warp	32.00 (+0.00%)
Warp Cycles Per Executed Instruction [cycle]	27.69 (-78.58%)	Avg. Not Predicated Off Threads Per Warp	26.99 (-2.43%)

CPI Stall 'Long Scoreboard' On average each warp of this kernel spends 21.3 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture) operation. This represents about 78.4% of the total average of 27.2 cycles between issuing two instructions. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality or by changing the cache configuration, and consider moving frequently used data to shared memory.

Warp Stall Warp stall analysis

Fig. 13. Profiling the HemeLB_GPU main collision-streaming kernel (*Inner domain & Walls*), running at double precision, using NVIDIA Nsight Compute on V100 GPU following optimisations applied at the UK National GPU Hackathon 2022 event. These involved mainly loop unrolling, merging loops and increasing the maximum number of registers per thread. (a) GPU utilization: Compute and memory throughput. (b) Roofline analysis: The arithmetic intensity, i.e. the ratio between compute work (FLOPs) and data movement (bytes), increased from 0.32 to 0.73. (c) Increasing the maximum number of registers per thread (compilation with the flag -maxrregcount 200) resulted in 87 registers (up from 40) and decreased the stalled warps by 82% (from 117.0 to 21.3 cycles), as the GPU made more use of local memory.

Acknowledgements

We acknowledge funding support from: (a) the European Commission CompBioMed Centre of Excellence (Grant No. 675451 and 823712), (b) the UK Engineering and Physical Sciences Research Council under the project 'UK Consortium on Mesoscale Engineering Sciences (UKCOMES)' (Grant No. EP/R029598/1), (c) the Medical Research Council (MRC) for a Medical Bioinformatics grant (MR/L016311/1) and (d) special funding from the UCL Provost.

We gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (www.lrz.de). This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk), as well as the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk). We acknowledge PRACE for awarding us access to JUWELS at GCS@FZJ, Germany and Piz Daint at CSCS, Switzerland. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

We would like to thank Brian J. N. Wylie (Jülich Supercomputing Centre & Performance Optimisation and Productivity Centre of Excellence in HPC) for profiling and analysing the performance of HemeLB_GPU on Juwels Cluster and Benjamín Hernández (Oak Ridge National Laboratory) for useful discussions and guidance on profiling and running simulations on Summit. We would also like to thank Paul Graham (NVIDIA) and Michel Herrera Sanchez (BNP Paribas CIB) for helpful discussions and support during the UK National GPU Hackathon 2022 event, which was partnered with UKRI, NVIDIA, and OpenACC.org.

References

[1] Y. Shi, P. Lawford, R. Hose, Biomed. Eng. Online 10 (1) (2011) 1–38, https://doi.org/10.1186/1475-925X-10-33.
[2] C. Sheng, S.N. Sarwal, K.C. Watts, A.E. Marble, Med. Biol. Eng. Comput. (ISSN 1741-0444) 33 (1) (Jan 1995) 8–17, https://doi.org/10.1007/BF02522938.
[3] M.U. Qureshi, G.D.A. Vaughan, C. Sainsbury, M. Johnson, C.S. Peskin, M.S. Olufsen, N.A. Hill, Biomech. Model. Mechanobiol. (ISSN 1617-7940) 13 (5) (Oct 2014) 1137–1154, https://doi.org/10.1007/s10237-014-0563-y.
[4] L.O. Müller, E.F. Toro, Int. J. Numer. Methods Biomed. Eng. 30 (7) (2014) 681–725, https://doi.org/10.1002/cnm.2622.
[5] J.P. Mynard, J.J. Smolich, Ann. Biomed. Eng. (ISSN 1573-9686) 43 (6) (2015) 1443–1460, https://doi.org/10.1007/s10439-015-1313-8.
[6] N. Xiao, J.D. Humphrey, C.A. Figueroa, J. Comput. Phys. (ISSN 0021-9991) 244 (2013) 22–40.
[7] A. Randles, E.W. Draeger, P.E. Bailey, J. Comput. Sci. (ISSN 1877-7503) 9 (2015) 70–75.
[8] J.W.S. McCullough, R.A. Richardson, A. Patronis, R. Halver, R. Marshall, M. Ruefenacht, B.J.N. Wylie, T. Odaker, M. Wiedemann, B. Lloyd, E. Neufeld, G. Sutmann, A. Skjellum, D. Kranzlmüller, P.V. Coveney, Interface Focus 11 (1) (2021) 20190119, https://doi.org/10.1098/rsfs.2019.0119.
[9] M.D. Mazzeo, P.V. Coveney, Comput. Phys. Commun. (ISSN 0010-4655) 178 (12) (2008) 894–914, https://doi.org/10.1016/j.cpc.2008.02.013.
[10] D. Groen, J. Hetherington, H.B. Carver, R.W. Nash, M.O. Bernabeu, P.V. Coveney, J. Comput. Sci. (ISSN 1877-7503) 4 (5) (2013) 412–422, https://doi.org/10.1016/j.jocs.2013.03.002.
[11] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M.B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, et al., Comput. Math. Appl. 81 (2021) 334–350, https://doi.org/10.1016/j.camwa.2020.03.022.
[12] C. Feichtinger, S. Donath, H. Köstler, J. Götz, U. Rüde, J. Comput. Sci. 2 (2) (2011) 105–112, https://doi.org/10.1016/j.jocs.2011.01.004.
[13] M. Bauer, S. Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnes, H. Köstler, R. Ulrich, Comput. Math. Appl. (ISSN 0898-1221) 81 (2021) 478–501, https://doi.org/10.1016/j.camwa.2020.01.007.
[14] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, E. Kaxiras, Concurr. Comput., Pract. Exp. 22 (1) (2010) 1–14, https://doi.org/10.1002/cpe.1466.
[15] M. Bernaschi, M. Bisson, M. Fatica, S. Melchionna, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1–11.
[16] A.P. Randles, V. Kale, J. Hammond, W. Gropp, E. Kaxiras, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IEEE, 2013, pp. 1063–1074.
[17] J. Latt, C. Coreixas, J. Beny, PLoS ONE 16 (4) (2021) 1–29, https://doi.org/10.1371/journal.pone.0250306.
[18] C. Obrecht, F. Kuznik, B. Tourancheau, J-J. Roux, Comput. Math. Appl. 65 (2) (2013) 252–261, https://doi.org/10.1016/j.camwa.2011.02.020.
[19] W. Xian, A. Takayuki, Parallel Comput. 37 (9) (2011) 521–535, https://doi.org/10.1016/j.parco.2011.02.007.
[20] E. Calore, A. Gabbana, J. Kraus, E. Pellegrini, S.F. Schifano, R. Tripiccione, Parallel Comput. 58 (2016) 1–24, https://doi.org/10.1016/j.parco.2016.08.005.
[21] P. Valero-Lara, J. Jansson, Concurr. Comput., Pract. Exp. 29 (7) (2017) e3919, https://doi.org/10.1002/cpe.3919.
[22] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, E. Kaxiras, Concurr. Comput., Pract. Exp. 22 (1) (2010) 1–14, https://doi.org/10.1002/cpe.1466.
[23] M. Bernaschi, S. Matsuoka, M. Bisson, M. Fatica, T. Endo, S. Melchionna, in: SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2011, pp. 1–12.
[24] S. Melchionna, M. Bernaschi, S. Succi, E. Kaxiras, F.J. Rybicki, D. Mitsouras, A.U. Coskun, C.L. Feldman, Comput. Phys. Commun. 181 (3) (2010) 462–472, https://doi.org/10.1016/j.cpc.2009.10.017.
[25] C. Huang, B. Shi, Z. Guo, Z. Chai, Commun. Comput. Phys. 17 (4) (2015) 960–974, https://doi.org/10.4208/cicp.2014.m342.
[26] K. Mattila, T. Puurtinen, J. Hyväluoma, R. Surmas, M. Myllys, T. Turpeinen, F. Robertsén, J. Westerholm, J. Timonen, J. Comput. Sci. 12 (2016) 62–76, https://doi.org/10.1016/j.jocs.2015.11.013.
[27] F. Robertsen, J. Westerholm, K. Mattila, in: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, IEEE, 2015, pp. 604–609.
[28] M.J. Mawson, A.J. Revell, Comput. Phys. Commun. 185 (10) (2014) 2566–2574, https://doi.org/10.1016/j.cpc.2014.06.003.
[29] N.P. Tran, M. Lee, S. Hong, Sci. Program. (2017) 2017, https://doi.org/10.1155/2017/1205892.
[30] G. Herschlag, S. Lee, J.S. Vetter, A. Randles, IEEE Trans. Parallel Distrib. Syst. 32 (10) (2021) 2400–2414, https://doi.org/10.1109/TPDS.2021.3061895.
[31] G. Herschlag, S. Lee, J.S. Vetter, A. Randles, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 825–834.
[32] B. Feiger, J. Gounley, D. Adler, J.A. Leopold, E.W. Draeger, R. Chaudhury, J. Ryan, G. Pathangey, K. Winarta, D. Frakes, F. Michor, A. Randles, Sci. Rep. 10 (1) (2020) 1–13, https://doi.org/10.1038/s41598-020-66225-0.
[33] C.A. Figueroa, C.A. Taylor, A.L. Marsden, in: Encyclopedia of Computational Mechanics, second edition, 2017, pp. 1–31.
[34] S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond, Oxford University Press, Oxford, 2001.
[35] A.A. Mohamad, Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes, Springer, London, 2011.
[36] Z. Guo, C. Shu, Lattice Boltzmann Method and Its Applications in Engineering, WORLD SCIENTIFIC, 2013.
[37] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, E.M. Viggen, The Lattice Boltzmann Method: Principles and Practice, Springer, 2017.
[38] S. Succi, The Lattice Boltzmann Equation: For Complex States of Flowing Matter, Oxford University Press, Oxford, 2018.
[39] P.L. Bhatnagar, E.P. Gross, M. Krook, Phys. Rev. 94 (3) (1954) 511, https://doi.org/10.1103/PhysRev.94.511.
[40] L. Luo, Phys. Rev. E 62 (Oct 2000) 4982–4996, https://doi.org/10.1103/PhysRevE.62.4982.
[41] A.J.C. Ladd, R. Verberg, J. Stat. Phys. 104 (5) (2001) 1191–1251, https://doi.org/10.1023/A:1010414013942.
[42] C. Pan, L.-S. Luo, C.T. Miller, Comput. Fluids 35 (8–9) (2006) 898–909, https://doi.org/10.1016/j.compfluid.2005.03.008.
[43] D. Groen, R.A. Richardson, R. Coy, U.D. Schiller, H. Chandrashekar, F. Robertson, P.V. Coveney, Front. Physiol. 9 (2018) 721, https://doi.org/10.3389/fphys.2018.00721.
[44] S.C.Y. Lo, J.W.S. McCullough, P.V. Coveney, Imposing ratios of outlet flow rates on large arterial networks with two-element Windkessel model: parametric analysis, https://doi.org/10.21203/rs.3.rs-1609811/v1, 2022.
[45] A. Ladd, J. Fluid Mech. 271 (1994) 285, https://doi.org/10.1017/S0022112094001771.
[46] R.W. Nash, H.B. Carver, M.O. Bernabeu, J. Hetherington, D.K. Groen, T. Krüger, P.V. Coveney, Phys. Rev. E 89 (2014) 023303, https://doi.org/10.1103/PhysRevE.89.023303.
[47] G. Wellein, T. Zeiser, G. Hager, S. Donath, Comput. Fluids 35 (8–9) (2006), https://doi.org/10.1016/j.compfluid.2005.02.008.
[48] B.J.N. Wylie, HemeLB_GPU on JUWELS/V100 performance assessment (POP2_AR_065), techreport POP AR 065, POP CoE/Jülich Supercomputing Centre, 2020.
[49] HemeLB_GPU github repository https://github.com/UCL-CCS/HemePure-GPU/, 2022.
[50] Running a HemeLB simulation, http://hemelb.org.s3-website.eu-west-2.amazonaws.com/tutorials/simulation/, Sep 2018.
[51] Top 500 - the List, https://www.top500.org/.
[52] Performance Optimisation and Productivity: a Centre of Excellence in HPC, https://pop-coe.eu/, 2021.
[53] B.J.N. Wylie, HemeLB performance assessment report, techreport POP AR 102, POP CoE/Jülich Supercomputing Centre, 2018, https://pop-coe.eu/sites/default/files/pop_files/pop-ar-hemelb-b.pdf.
[54] B.J.N. Wylie, HemeLB on SuperMUC-NG performance assessment report, techreport POP2 AR 041, POP CoE/Jülich Supercomputing Centre, 2020, https://pop-coe.eu/sites/default/files/pop_files/pop2-ar-hemelb.pdf.
[55] W. Joubert, Summit Node Bandwidths: Performance Results, techreport, Scientific Computing Group, Oak Ridge National Laboratory, 2019, https://www.olcf.ornl.gov/wp-content/uploads/2019/02/STW_Feb_2019-02-SummitNodePerformance-WJ.pdf.
[56] E.G. Boman, Ü.V. Çatalyürek, C. Chevalier, K.D. Devine, Sci. Program. 20 (2) (2012) 129–150, https://doi.org/10.3233/SPR-2012-0342.
[57] G. Karypis, METIS: Unstructured graph partitioning and sparse matrix ordering system, Technical report, 1997.
[58] M. Schönherr, K. Kucher, M. Geier, M. Stiebler, S. Freudiger, M. Krafczyk, Comput. Math. Appl. (ISSN 0898-1221) 61 (12) (2011) 3730–3743, https://doi.org/10.1016/j.camwa.2011.04.012.
[59] S. Jha, RADICAL cybertools, online, https://radical-cybertools.github.io/, 2021.
[60] E. Mayer, S. Cielo, J.W.S. McCullough, J. Gunther, P.V. Coveney, Visualization of human-scale blood flow simulation using Intel OSPRay Studio on SuperMUC-NG, online, https://www.oneapi.io/event-sessions/visualization-of-human-scale-blood-flow-tech-talk/, June 2021.
[61] HemePure Tools repository, URL https://github.com/UCL-CCS/HemePure_tools.
[62] hemeXtract tool to convert the output files to human readable format, URL https://github.com/UCL-CCS/hemeXtract.
[63] E. Calore, A. Gabbana, S.F. Schifano, R. Tripiccione, Int. J. High Perform. Comput. Appl. 33 (1) (2019) 124–139, https://doi.org/10.1177/1094342017703771.
[64] N. Weber, M. Goesele, in: EGPGV@ EuroVis, 2014, pp. 57–64.
[65] K. Kofler, B. Cosenza, T. Fahringer, in: European Conference on Parallel Processing, Springer, 2015, pp. 263–274.
[66] NVIDIA Nsight Compute, URL https://developer.nvidia.com/nsight-compute.
[67] NVIDIA Nsight Systems, URL https://developer.nvidia.com/nsight-systems.
[68] J. Habich, C. Feichtinger, H. Köstler, G. Hager, G. Wellein, Comput. Fluids 80 (2013) 276–282, https://doi.org/10.1016/j.compfluid.2012.02.013.
[69] P.R. Rinaldi, E.A. Dari, M.J. Vénere, A. Clausse, Simul. Model. Pract. Theory 25 (2012) 163–171, https://doi.org/10.1016/j.simpat.2012.03.004.

[70] J.-M. Zhang, L. Zhong, T. Luo, Y. Huo, S.Y. Tan, A.S.L. Wong, B. Su, M. Wan, X. Zhao, G.S. Kassab, et al., BioMed Res. Int. 2014 (2014), https://doi.org/10.1155/2014/514729.
[71] N.M. Maurits, G.E. Loots, A.E.P. Veldman, J. Biomech. 40 (2) (2007) 427–436, https://doi.org/10.1016/j.jbiomech.2005.12.008.
[72] J.W.S. McCullough, P.V. Coveney, Sci. Rep. 11 (1) (2021) 1–11, https://doi.org/10.1038/s41598-021-03584-2.
[73] J.W.S. McCullough, P.V. Coveney, Sci. Rep. 11 (1) (2021) 1–12, https://doi.org/10.1038/s41598-021-01435-8.