International Journal of Parallel, Emergent and Distributed Systems

ISSN: (Print) (Online) Journal homepage: www.tandfonline.com/journals/gpaa20

3D multicore CPU vs GPU on sparse patterns of Sleptsov net virtual machine

D. A. Zaitsev, Y. Ajima, J. F. C. Bartlett & A. Kumar

To cite this article: D. A. Zaitsev, Y. Ajima, J. F. C. Bartlett & A. Kumar (16 Apr 2025): 3D multicore CPU vs GPU on sparse patterns of Sleptsov net virtual machine, International Journal of Parallel, Emergent and Distributed Systems, DOI: 10.1080/17445760.2025.2490148

To link to this article: https://doi.org/10.1080/17445760.2025.2490148

Published online: 16 Apr 2025.

INTERNATIONAL JOURNAL OF PARALLEL, EMERGENT AND DISTRIBUTED SYSTEMS
https://doi.org/10.1080/17445760.2025.2490148

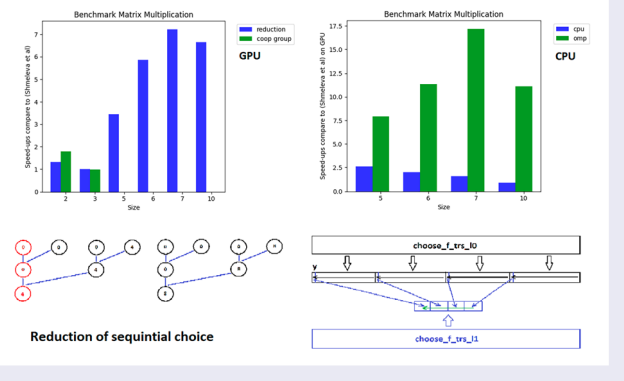
3D multicore CPU vs GPU on sparse patterns of Sleptsov net virtual machine

D. A. Zaitsev [a], Y. Ajima [b], J. F. C. Bartlett [a,c] and A. Kumar [a,c]

[a]School of Computing, University of Derby, Derby, UK ; [b]Advanced Technology Development Unit, Fujitsu Limited, Kawasaki-shi, Japan; [c]Rolls-Royce, Derby, UK

ABSTRACT

A Sleptsov net is a discrete-event system, capable of universal computations, applied as a graphical language of concurrent programming for HPC and embedded domains. Search for the first fireable transition in a sequence of transitions, reordered by their priorities, represents a challenge for mass-parallel devices and a bottleneck of the virtual machine. A reduction technique for sequential search implementation on GPU, with less than logarithmic time complexity concerning the sequence length, has been developed. Together with the conventional reduction of minimum and cooperative groups, the techniques yield about ten times speed-up on CPUs and GPUs.



ARTICLE HISTORY
Received 25 March 2025
Accepted 3 April 2025

KEYWORDS
HPC; GPU; Sleptsov net computing; virtual machine; reduction of sequential choice; collective groups

1. Introduction

Sleptsov net computing [1] mends imperfections of modern HPC architecture [2] and provides a toolset for reliable embedded system design [3], expanding to other domains as mentioned in [4]. In view of future dedicated hardware implementations of Sleptsov net (SN) computer [5], we offer a rather good compromise using conventional devices with mass-parallel computing facilities such as GPUs and FPGAs. In the present paper, observing recent advances in the application of FPGAs for Sleptsov net based reliable embedded system design [3], we focus on using GPUs.

The SN virtual machine (VM) for GPU, described in [6], has been recently refined with a considerable speed-up because of using an ad-hoc format of a sparse matrix, called a matrix with condensed

CONTACT D. A. Zaitsev daze@acm.org School of Computing, University of Derby, Kedleston Road, Derby DE22 1GB, UK

columns (MCC) and the transition reordering based on their priorities combined with the first fireable transition choice, developed in [7]. The SN VM [7] supports compatibility with NVIDIA GPU architecture 35 and uses a multiple kernel implementation of an SN step on a GPU.

The general theory of discrete-event systems (DES) simulation [8, 9] covers issues of using parallel and distributed systems, which are further specialized for mass-parallel devices, such as GPUs, in [10–14], in particular, for Petri nets [15–17]. However, the cited papers offer neither detailed specification of solutions nor open source implementations and use rather tiny nets for runs which occupy milliseconds. Besides, SN VMs possess a series of peculiarities compared with both the simulation and the calculation of state space for DES in general and, in particular, for the variety of place-transition (Petri) nets. We use multiple firing of a transition at a step and we are interested in computing, as fast as possible, a single trajectory, the way of the firing transition choice does not matter because we are interested in the results of computation on SNs, which are invariant with respect to the firing transition sequence. We develop efficient ways of using mass-parallel computing resources for sequential DESs based on special forms [3, 7] of sparse data [18, 19] representation.

There are certain difficulties in using conventional mass-parallel computing devices, such as GPUs, for fast simulation of DESs represented by process algebra [20], multiset rewriting systems [21, 22], place-transition nets [23] and other formal systems. Thus, the present study application domain is considerably wider than Sleptsov nets. The central problem is represented by the fact that the DES behaviour bears a sequential character specified as a sequence of steps transforming the DES state. Having no possibility of modifying the corresponding definitions, we can, at first, parallelize a step implementation using the maximal required number of GPU [24] threads having the appropriate spatial structure and, at second, implement the loop, that controls the DES sequence of steps, within a GPU kernel. The second approach was limited by a single GPU block (up to 1024 threads) in early NVIDIA GPU architecture. With the cooperative group facility [25, 26], we can implement a global synchronization of the entire grid, which we employ in the present paper.

The purpose of the present paper is to explore modern NVIDIA GPU architecture features [24], in particular, cooperative group facilities, yielding a possibility of global synchronization over GPU blocks, to further speed up SN VM for GPU. We also study the influence of the reduction [27, 28] application on the VM performance, for both a conventional reduction of minimum and a specially designed reduction of sequential choice that has definite prospects in the DESs behaviour implementation on GPU. The results are also compared to SN VMs designed for multicore CPUs, which can compete with GPUs in view of DESs' sequential behaviour.

2. Sleptsov net computing overview

A concept of multiple firing of a transition at a step (a multi-channel transition) introduced in [29] for a timed Petri net has gained its further development for a nontimed place-transition net [1] which was called a Sleptsov net. The new term is justified by the fact that an SN runs exponentially faster than a PN [1] and represents a Turing-complete system [30, 31]. Recently, many scientists have used a similar technique to speed up multiset rewriting systems, spiking P neuron systems, DNA computing, etc., referring to it as an 'exhaustive use of rule'.

2.1. Sleptsov net concept

A *Sleptsov net* is a bipartite directed graph supplied with a dynamic process. One part of vertexes depicted as circles (ovals) is called *places*. The other part of vertexes depicted as squares (rectangles) is called *transitions*. Inside places, dynamic elements called *tokens* are situated. The net runs in discrete time divided in equal intervals called steps. The net state, represented via the distribution of tokens over places and called a *marking*, changes in the moment of the step change as a result of *transition firing*. We consider also the *arc multiplicity* as a natural number. We call a *transition incoming arc firing*

multiplicity the marking of its input place divided by the arc multiplicity; we call a t*ransition firing multiplicity* the minimal firing multiplicity over its incoming arcs. A transition with firing multiplicity greater than zero is called a *fireable transition*. In SN, a single fireable transition fires in its maximal firing multiplicity. At a step, any fireable transition can fire which makes the net behaviour nondeterministic. Note that a Petri net [32] represents a special case of a Sleptsov net.

Though Sleptsov nets are Turing-complete [30, 31], it is not convenient to compose programs using them. Because of this fact, we use additional types of arcs – inhibitor and priority – introduced by Agerwala and Hack, respectively [33]. An *inhibitor arc* is directed from a place to a transition; we consider it to have an infinite firing multiplicity when the place marking equals zero and zero firing multiplicity otherwise. The inhibitor arc has a hollow small circle at its end. A *priority arc* connects two transitions; we consider the transition from where the arc starts has higher priority than the transition where the arc ends. It means the lower priority transition does not fire when the higher priority transition is fireable.

Let us formalize the description of SNs and their dynamics based on their verbal definitions. An SN is a tuple $N = (P, T, A, R, \mu)$, where $P = \{p\}$ is a finite set of places, $T = \{t\}$ is a finite set of transitions, a mapping $A : (P \times T) \cup (T \times P) \to \mathbb{Z}^{\geq -1}$ defines arcs, connecting places and transitions, their type and multiplicity, a partial order relation $R \subset (T \times T)$ specifies priorities of transitions, a mapping $\mu : P \to \mathbb{Z}^{\geq 0}$ represents marking of places, μ_0 is its initial value; a place marking is depicted as a number inside the place or as the required number of dots. Mapping A produces either: a zero value that means the absence of the corresponding arc, a natural value that means a regular arc of specified multiplicity, or -1 that denotes an inhibitor arc, this value is valid only for arcs connecting places with transitions.

Let us specify the *transition input arc fireability multiplicity* as (1) and the *transition firing multiplicity* as (2). Any *fireable transition* ($C(t) > 0$) *fires* at a step in $C(t)$ instances, changing the net marking according to (3).

$$C(p,t) = \begin{cases} \mu(p)/A(p,t), & \text{if } A(p,t) > 0 \\ \infty, & \text{if } A(p,t) < 0 \wedge \mu(p) = 0 \\ 0, & \textit{otherwise} \end{cases} \tag{1}$$

$$C(t) = \min_{A(p,t) \neq 0} (C(p,t)) \tag{2}$$

$$\mu(p) = \mu(p) - C(t) \cdot A(p,t) + C(t) \cdot A(t,p), A(p,t) \neq -1 \tag{3}$$

For processing by computer, we specify an SN as:

- a pair of numbers m and n, where $m = |P|$ and $n = |T|$;
- a pair of matrices of incoming B and outgoing D arcs of transitions, respectively, where $b(p,t) = A(p,t)$, $d(p,t) = A(t,p)$;
- a matrix of priority relation R, which, in the majority of cases, is more convenient to replace by its transitive closure R';
- a vector of marking mu, where $mu(p) = \mu(p)$.

When studying SN properties [33], we consider the net complete behaviour which is nondeterministic and, in the general case, infinite. Basically, three groups of techniques are applicable: state space analysis, including finite representation of state space and recent symbolic techniques; linear algebra methods of solving systems of algebraic equations and inequalities; and reduction methods to reduce the net size preserving its properties. From a practical point of view, we are most interested in such properties as boundedness and liveness; liveness can be formulated in a simplified form of deadlock absence. Thus, with SN programming we have a wide spectrum of formal methods for concurrent software verification.

2.2. Sleptsov net computing IDE

Using an SN as a concurrent graphical programming language possesses a series of advantages: fine granulation of concurrent processes, uniform graphical language, the possibility of computing memory implementation (without processor-memory bottleneck), and wide applicability of formal methods for concurrent software verification. Note that compared to a PN, an SN runs exponentially faster and is Turing-complete. In the early published papers [1, 5, 34], we developed technology of programming in SNs language and the corresponding IDE [6]. We consider plain (low-level) inhibitor, priority Sleptsov nets as a machine language implemented either by the corresponding dedicated hardware (prospect) or by a virtual machine.

For composing programs, we offer high-level SNs specified based on separating places to store variables, using the transition substitution by a subnet for the hierarchical composition of programs, and an abbreviation of memory copying and movement operations represented by dashed arcs. At first, we develop basis subnets that implement arithmetic and logic operations. Then, we develop rules of sequential and nested program composition. We also specify nets corresponding to basic statements of a programming language: branching, loops, and parallel execution. We offer a control flow, data flow, combined, and arbitrary approaches for the program composition. Note that, especially when using extended Sleptsow-Salwicki transition firing rules, we stop perceiving a sequential way of thinking encouraged by a sequence of statements of a textual programming language. In the process of software design, the main goal is to preserve the original concurrency of the application domain. All SN transitions are concurrent initially, if we imagine them isolated from places, like actions. Then, using places and arcs, we apply minimal possible restrictions on the initial maximal concurrency. Thus, there is no need for subsequent parallelization of developed software. If Sleptsov net machines can execute all the transitions in parallel, we achieve the maximal possible efficiency of a concurrent run having certain restrictions ensured by a definite SN.

In Figure 1(a), we represent an SN program composed based on the control flow approach for RSA encryption/decryption while in Figure 1(b), we show an SN program for mass-parallel solving of the Laplace equation based on the data flow approach.

Though SN behaviour is a non-deterministic one, when there are alternatives in the firing transition choice, in the process of an SN program run, we consider a single trace satisfying the transition firing rules having proven the fact [1] that the computation result is invariant with respect to the valid trace choice. The invariance is proven for initial nets and the program composition rules.

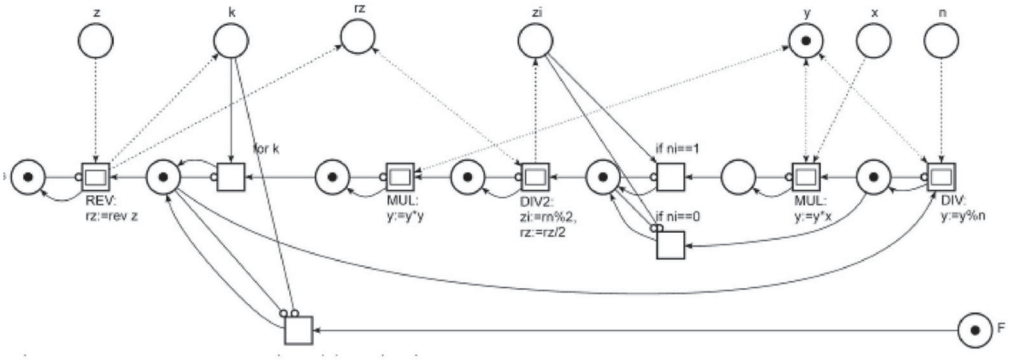
An SN IDE (Figure 2) has been presented in [6]. We use Tina toolset [35] as a graphical editor for drawing SN programs and specifying the transition substitution for hierarchical design, applying a wide spectrum of formal methods for software verification. We developed a dedicated compiler-linker of SNs for hierarchical design and SN virtual machines implemented on CPU and GPU, to run SN programs including recent updates [3, 7].

2.3. Sleptsov net virtual machine

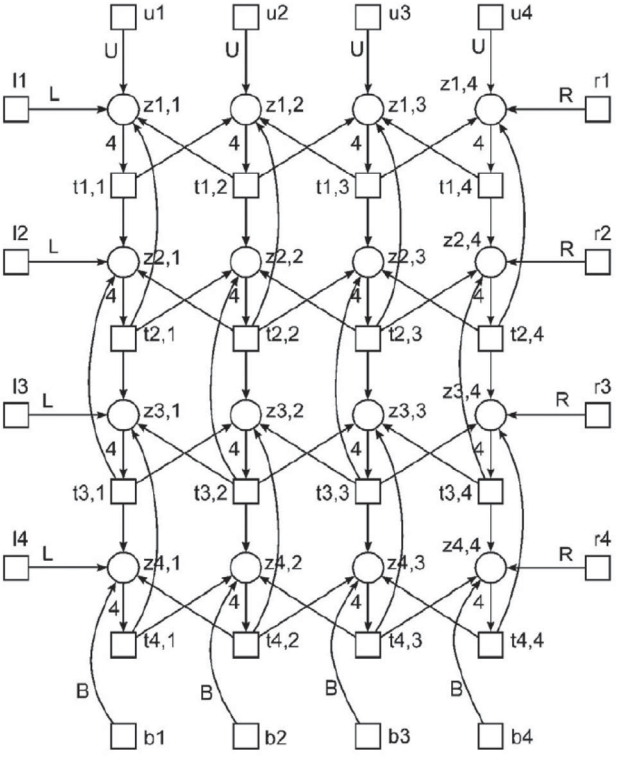
The basic algorithm of SN VM work represents processes of computing the fireability conditions and firing a chosen transition changing the current marking according to (1) – (3). The best-known implementation [7], uses the corresponding stages (kernels) extended with an additional stage (kernel) for the firing transition choice. Thus, we have the four following stages:

(1) compute the firing multiplicity of transition incoming arcs;
(2) compute the firing multiplicity of transitions;
(3) choose a firing transition;
(4) fire the chosen transition calculating the next marking.

In the present paper, we are going to implement this algorithm as fast as possible using the reduction and global synchronization based on CUDA cooperative groups.



(a) RSA encryption/decryption – control flow approach, transition substitution;



(b) numerical solving Laplace equation – data flow approach.

Figure 1. Examples of SN programs [1]. (a) RSA encryption/decryption – control flow approach, transition substitution. (b) numerical solving Laplace equation – data flow approach.

We specify the algorithm with a scheme shown in Figure 3 using the mass-parallel algorithm notation introduced in [36]. Following [36], we depict a loop by a section, a double nested loop – by a rectangle, a triple nested loop – by a cube, etc. There is a rather straightforward corresponding of stages I, II, and IV to the formulae (1) – (3), where function `m_fire(p,t)` implements formula (1), function `RED_min(y,t)` implements formula (2), and function `new_mu(mu,ft,fc)` implements formula (3). We use an auxiliary matrix *y* and a vector *z* to store the firing multiplicities of arcs and transitions, respectively.

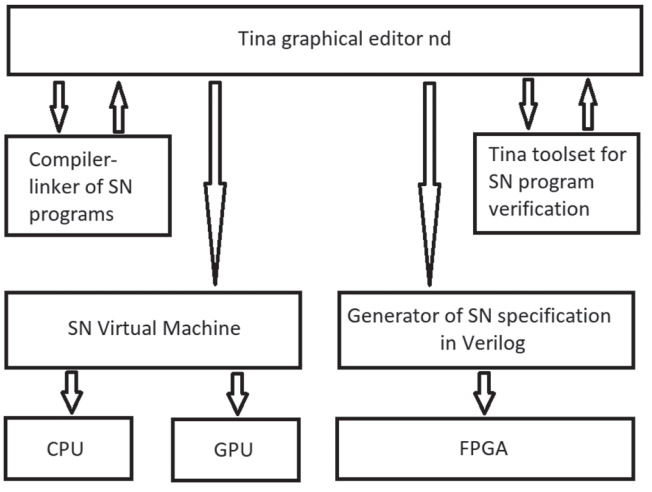


Figure 2. SNC IDE.

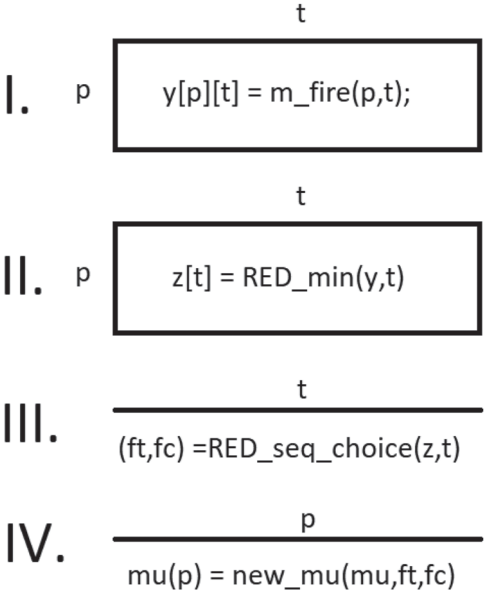


Figure 3. SN VM step mass-parallel chart.

Here we observe possibilities of applying reduction [27, 28] to enhance VM performance compared [36] that is reflected in the chart. At first (for stage II), we apply a reduction of a binary operation, in particular, minimum. At second (for stage III), we apply our ad-hoc reduction of sequential choice, which we develop in the present paper. We believe it will find wide application in simulating DES. Thus, we employ squares and vectors of threads, obtaining an additional speed-up having, at stage II, a logarithmic time complexity, instead of the linear complexity, in *mm* and, at stage III, a logarithmic time complexity instead of the linear complexity in *t*, compared to VM described in [36]. Besides, for cooperative groups, supported by recent CUDA architecture, because of using global thread synchronization over the entire grid, we implement a single kernel with a loop over the SN time (a sequence of steps) and the sequence of stages I–IV, working on a square grid of size $mm \times t$.

Note that actually, having the sparse matrix format MCC [36], we work with a grid of size $mm \times n$ instead of $m \times n$ where mm is the maximal number of nonzero elements in a column over matrices B and D which specify SN arcs. For nets, which represent SN programs, the magnitude mm is rather small compared to m and tends to not grow with the growing net size. We omit matrix R because of the transition reordering based on their priorities. We represent each matrix by a pair of matrices with the following suffices: `_v` containing values of nonzero elements and `_i` containing indexes of nonzero elements. Such a format is a certain compromise between zero-free row- or column- wide sparse representation [3] and the convenience of the conventional matrix form for implementation on GPU. Padding rather a modest number of zeroes, we have an overhead of only one indirect access via the nonzero element index $p = B\_i[pi][t]$ that is optimized for processing incoming and outgoing arcs of transitions.

3. Reduction of sequential choice

We speed up the solution of the following task on a mass parallel computing device, such as a GPU. For a given array z over the set $\{0, 1\}$, find the first in the sequence element (its index) which equals to the unit (does not equal to zero for an extended set of nonnegative integer numbers). We apply the interleaving reduction approach supplied with an early break when the zero element becomes nonzero, illustrated in Figure 4 for the best, Figure 4(a), an average, Figure 4(b), and the worst, Figure 4(c), cases.

Before the reduction loop, we replace the units with the element index plus one to have a nonzero representation of indexes. In the first passage, using $n/2$ threads, we process elements pairwise, replacing the first element in the pair with the second element, in case the first element equals zero and the second one is greater than zero. Then, we continue in the same way on the obtained array, increasing the step by 2. In case the first element of the array (with zero index) becomes nonzero, we break the process. For processing an array over natural numbers instead of set $\{0, 1\}$, we process an auxiliary array of the same size which stores indexes of nonzero elements of the source array. The algorithm is represented in Figure 5.

Statement 1. The algorithm shown in Figure 5 finds the first nonzero element in the sequence.

Proof: If the first element (with zero index) is nonzero at any passage of the loop, we have found it because of the break. In the first loop passage, if within each pair there is a nonzero element, it becomes the leftmost, its actual index preserved. In the second passage, the first nonzero element becomes the leftmost within each four elements, and so on. For the array lengths that are not a power of two, the index check prevents exceeding the memory bounds. ■

Statement 2. The algorithm's, shown in Figure 5, worst time complexity is $O(\log_2 lz)$ and average time complexity is $O(\log_2 lz/(nnz + 1))$, where lz is the array length and nnz is the number of nonzero elements.

Proof: In the worst case, we do not break the loop, thus it is repeated $log_2\ lz$ times. For an average case, we suppose that nnz nonzero elements are spread uniformly within the array z dividing it into $nnz + 1$ stretches of zero elements, approximate size of each stretch is $lz/(nnz + 1)$. Thus, an average distance to the first nonzero element is evaluated as $lz/(nnz + 1)$. It requires approximately $O(\log_2\ lz/(nnz + 1))$ passages of the loop to process the corresponding sub-array. ■

4. Architectural trends and performance characteristics of GPUs and CPUs

This section discusses the main features and progress directions in the architectures of GPUs and CPUs in recent years, as well as their performance characteristics for Sparse Matrix-Vector Multiplication (SpMV), which is the core algorithm of SN VM, considering the matrix form of the SN state equation similar to [33].

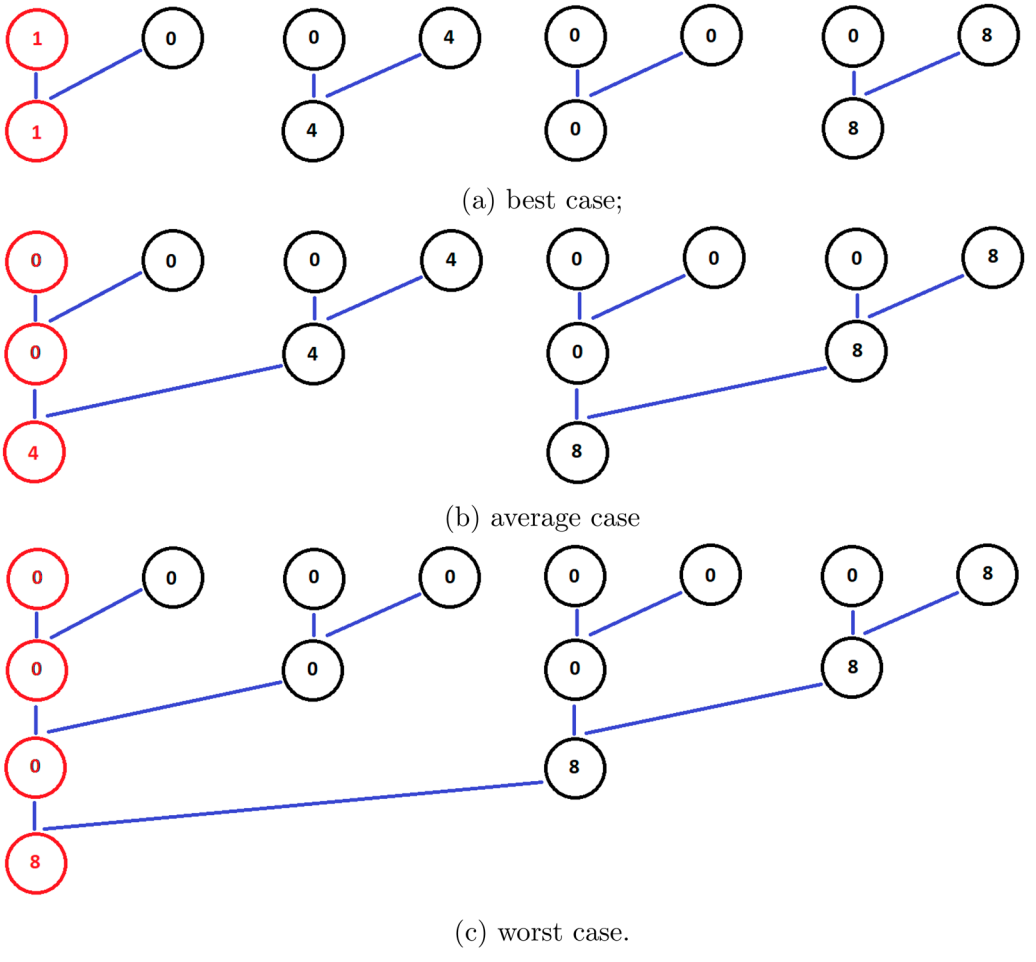


Figure 4. Examples of reduction of sequential choice. (a) best case. (b) average case. (c) worst case.

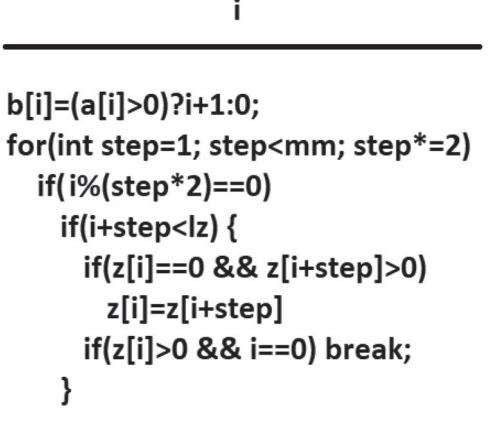


Figure 5. Mass-parallel chart for reduction of sequential choice.

4.1. Overview and key features of architectures

While GPUs and CPUs share similar core components, their architectures differ significantly. GPUs are equipped with thousands or even tens of thousands of very simple cores, making them suitable for tasks that handle large amounts of data requiring high parallelism. They are used as co-processors or accelerators. On the other hand, CPUs have dozens or over a hundred complex cores with high sequential processing capabilities. They are equipped with many specialized features optimized for low-latency sequential task processing, such as large-capacity cache memory, branch prediction, and out-of-order execution. It is expected that CPUs will achieve high performance in general-purpose applications, typical of binary-provided software and not optimized for specific systems.

Recent GPUs continue to innovate their architectures designed for HPC and AI workloads. They are equipped with tensor cores that have high computational throughput and HBM memory with high memory bandwidth, which streamlines large-scale matrix processing. Furthermore, the adoption of NVLink and Infinity Fabric, which accelerate communication between multiple GPUs, improves scalability in multi (4 or 8) socket configurations.

On the other hand, recent CPUs have memory channels capable of connecting large-capacity memory modules to meet the needs of data centres, and they continue to expand the number of cores and memory channels. The number of cores per socket has increased from tens to over a hundred, and the number of memory channels per socket has increased from 8 to 12. In addition, in multi (dual, and rarely quad) socket configurations of CPUs, the CPUs are connected by a special interconnect that maintains cache coherency, allowing software to transparently utilize cores from different sockets.

4.2. Performance characteristics and bottleneck factors of SpMV

SpMV is a crucial kernel not only in SN VM but also in graph processing and other HPC applications, and its performance significantly impacts the overall system efficiency. Understanding the performance characteristics of SpMV is essential for evaluating the architectures of GPUs and CPUs. Both sequential processing performance and global bandwidth are important for SpMV performance, and once sequential processing is sufficiently optimized, global bandwidth typically becomes the bottleneck.

In single-node systems, this global bandwidth bottleneck arises from the bandwidth of the memory or the inter-socket connection. In contrast, in multi-node systems, the bandwidth of the inter-node interconnection network becomes the bottleneck. SpMV presents unique performance challenges due to its data dependencies and irregular memory access patterns. As a result, various factors such as memory bandwidth intensity, low instruction-level parallelism (ILP), load imbalance, and memory latency overhead affect performance. The challenge in optimizing SpMV on different architectures is complicated by these factors and the trade-offs associated with sparse matrix storage formats. The SpMV algorithm, with its data-dependent and irregular memory access patterns, serves as an excellent benchmark for evaluating the memory subsystem and interconnect capabilities of HPC architectures.

4.3. Analysis of SpMV performance in the Graph500 benchmark

The Graph500 benchmark [37] is a ranking of supercomputers based on large-scale graph analysis, with SpMV being a central component. Results from Graph500 indicate that single-node execution of SpMV tends to have higher execution efficiency than multi-node execution, but there are limitations on the problem size. The maximum problem size for single-node execution is approximately 2^{37} vertices, whereas it is approximately 2^{43} vertices for multi-node execution.

Comparing the performance characteristics of GPUs and CPUs for SpMV in a single node, CPUs with high sequential processing performance have an advantage in naive implementations, but GPUs with

high memory bandwidth can be advantageous in optimized implementations. Depending on the data movement required when repeating SpMV, the bandwidth of the inter-socket interconnection network within a single node may become a bottleneck, potentially disadvantaging GPU nodes with high data movement. However, if there is a high-speed dedicated interconnect such as NVLink between GPU sockets and it is effectively utilized, GPU nodes may also have an advantage. The results of multi-node Graph500 show that the inter-node interconnection network plays a crucial role in performance scalability. The Graph500 results clearly demonstrate the trade-off between single-node efficiency and the need for multi-node systems to solve larger problems. They also suggest that the performance difference between CPUs and GPUs in SpMV varies depending on the level of optimization and specific hardware characteristics.

4.4. Supercomputer Fugaku and next-generation CPU FUJITSU-MONAKA

The supercomputer Fugaku [38], which has registered the world's highest performance in the Graph500 benchmark, is a representative example of an HPC system, employing the A64FX CPU with 48 cores per socket. The A64FX CPU is equipped with HBM for memory, similar to GPUs, and achieves high SpMV performance not only in naive implementations but also in optimized ones. Fugaku is configured with one socket per node, and Tofu Interconnect D [39] is used for inter-node communication. TofuD not only has high injection bandwidth per socket but also achieves extremely high bisection bandwidth across the entire system due to its exceptional scalability, connecting over 100,000 low-power A64FX CPUs, contributing to its Graph500 benchmark performance.

FUJITSU-MONAKA [38, 40, 41], scheduled for shipment in 2027, is the successor to the A64FX CPU. The main features of MONAKA include 144 cores, a dual-socket configuration, chiplet technology, three-dimensional stacking of the core die and SRAM die, and 2.5-dimensional implementation of four 3D-stacked units and one IO die. This state-of-the-art packaging technology significantly increases the number of high-performance cores. MONAKA is a CPU designed with the goal of applying HPC technology to a wide range of fields, and it will connect large-capacity memory modules instead of HBM for memory.

5. Enhanced SN machine for GPU

In this section, we focus on practical aspects of VM modification illustrated with snippets of code. We pursue not only a narrow purpose of describing our version of VM but also a wider one for enlightening readers on how to use the global synchronization and reduction [42, 43] to speed up their software. While the reduction of minimum is rather well-studied as a standard approach to speed-up computing a binary function over an array of data, the reduction of sequential choice contains some peculiarities, for instance, preliminary exit that makes its average time complexity lesser than logarithmic in case we consider the number of nonzero elements of the array as a parameter.

5.1. Applying reduction of minimum

After computing the firing multiplicity on all the incoming arcs of transitions employing all the threads of $mm \times t$ rectangle, we compute the firing multiplicity of transitions reducing with minimum the previous result over the incoming arcs, in particular on the first dimension (over columns) represented with parameter mm, corresponding to the number of employed threads within a block. Actually, we start with using $mm/2$ threads dividing the number by 2 on each next passage of the reduction loop, represented with the following code snippet:

```
int t = blockIdx.x;
int pi = threadIdx.x;
```

```
for(int step=1; step<mm; step*=2) {
    if(pi%(step*2)==0)
        if(pi+step<mm) y[pi][t] = min( y[pi][t], y[pi+step] );
    __syncthreads();
}
```

Macros *min()* computes the minimum of its two arguments. Each column corresponds to an SN transition and is stored on a separate GPU block. The code rewrites matrix *y*, the result is obtained in the first row of the matrix. For a small maximal number of the transition incoming/outgoing arcs, represented with parameter *mm*, we foresee some possibilities for additional speed-up when unrolling the loop on a GPU warp and packing a few transitions into a GPU block. Interleaving stages I and II of the step algorithm (Figure 3) seems inappropriate, yielding a certain slowdown because of additional synchronization.

5.2. Applying reduction of sequential choice

Reduction of sequential choice, to get the first fireable transition in the sequence, requires a more sophisticated approach because of a considerably greater number *n* of transitions than the maximal number *mm* among the transition incoming or outgoing arcs. Using a single thread of each block seems rather inefficient from the point of view of the warp structure of the GPU streaming multiprocessors, which are represented by threads of a block. Thus, we divide the array of size *n*, actually the first line of the auxiliary matrix *y*, into sections of size *b*2, which are implemented on a single block each and employ a two-level reduction that works for $n \leq 2^{20}$, which can be easily generalized for a multilevel variant. We use the following specification of blocks and the reduction calls for the multiple kernel implementation:

```
int b2=max_blk_size;
int g2=(n%max_blk_size==0)? n/max_blk_size: n/max_blk_size+1;
dim3 grid20  (g2);
dim3 block20 (b2);
dim3 grid21  (1);
dim3 block21 (g2);
choose_f_trs_l0<<<grid20, block20>>>(mm, n, d_y, dbg);
if(g2>1) choose_f_trs_l1<<<grid21, block21>>>(mm, n, d_y, b2, dbg);
```

The interaction of two kernels, which implement two-level reduction of sequential choice, is illustrated in Figure 6. Note that the kernels interact only via using common data, actually two first rows of matrix *y*. The array represented in blue is not actually copied, the corresponding elements of the source vectors *y*[0] and *y*[1] are accessed using the step equal to the partition size. The kernel *choose_f_trs_l0* reduces sequential choice within each partition of size *max_blk_size* of the firing multiplicity vector represented by the two first (with indexes 0 and 1) rows of matrix *y*. Its code snippet follows:

```
// choose_f_trs_l0:
unsigned int t = blockIdx.x*blockDim.x + threadIdx.x;
unsigned int tid = threadIdx.x;
unsigned int tmax = (blockIdx.x+1)*blockDim.x;
int step, goon=1;
y[1][t]=t;
for(step=1; (step<n)&goon; step*=2) {
    if(tid%(step*2)==0) {
        if(y[0][t]>0) {
```

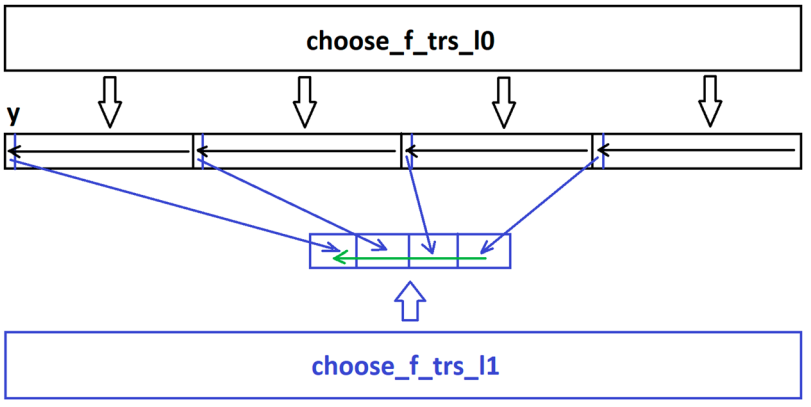


Figure 6. An example of two-level reduction of sequential choice work on four partitions of vector *y*[0].

```
            if(tid==0) goon=0;
        } else if((t+step<tmax)&(t+step<n)){
           if(y[0][t+step]>0)) {
              y[0][t]=y[0][t+step];
              y[1][t]=y[1][t+step];
              if(tid==0) goon=0;
           }
       }
    }
    __syncthreads();
}
```

Thus, *choose_f_trs_l0* processes each partition of *b*2 transitions on a separate GPU block, leaving the first found nonzero element at the beginning of it. Before the main loop, it copies the global transition number into the second row of the matrix *y*; remember that its first row contains the transition firing multiplicity. Variable *tid* represents a relative number of a thread within the current block to implement the break condition using variable *goon*. After *choose_f_trs_l0* finishes, the first element of each partition contains a specification of the first nonzero element within the partition: the element value – in the row with index zero and its actual index – in the row with index one.

The kernel *choose_f_trs_l1* reduces the sequential choice over the previously processed partitions of size *max_blk_size* of the firing multiplicity vector represented by the two first (with indexes 0 and 1) rows of matrix *y*. Its code snippet follows:

```
// choose_f_trs_l1:
unsigned int t = threadIdx.x*b2;
unsigned int tid = threadIdx.x;
int step, goon=1;
for(step=1; (step<n)&goon; step*=2) {
    if(tid%(step*2)==0) {
        if(y[0][t]>0) {
            if(tid==0) goon=0;
        } else if(t+step*b2<n){
          if((y[0][t+step*b2]>0)) {
              y[0][t]=y[0][t+step*b2];
              y[1][t]=y[1][t+step*b2];
```

```
                if(tid==0) goon=0;
            }
        }
    }
    __syncthreads();
}
```

Code *choose_f_trs_l1* resembles *choose_f_trs_l0* with the only change of the indexes *t* and *tid* specification, and using $step * b2$ for the next element offset. When it finishes, the result is contained in the first column of matrix *y*, namely in *y*[0][0] – the firing multiplicity and in *y*[1][0] – the firing transition number.

Note that two levels of the reduction of sequential choice implementation can be united into one kernel which is called sequentially with different parameters. The basic difference concerns how we treat the number of blocks within the grid and the number of threads within a block as well as the corresponding numbers of the current block and current thread. A multilevel reduction can be implemented using a more sophisticated kernel.

5.3. Composing a single kernel with global synchronization

Global synchronization over groups of blocks, an entire GPU grid, or a few GPUs in a cluster is provided with Cooperative Groups facility available starting from CUDA 9.0 [26]. To use Cooperative Groups classes and methods, we include the following headers and start working with the Cooperative Groups namespace:

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <cooperative_groups.h>
#include <cuda_runtime_api.h>
using namespace cooperative_groups;
```

To check Cooperative Groups support by the actual GPU architecture, we call function *cudaDeviceGetAttribute* after selecting a device with a function *cudaSetDevice* in the following way:

```
cudaSetDevice(0);
int supportsCoopLaunch = 0;
cudaDeviceGetAttribute(&supportsCoopLaunch,
cudaDevAttrCooperativeLaunch, DEV);
if(!supportsCoopLaunch) {
    printf("** error: no hardware support for Cooperative
    Launch\n");
    exit(ERR_NO_COOP_LAUNCH);
}
```

For the purpose of global synchronization over the entire grid, we create an object *grid*, which specifies the entire GPU grid including all blocks. Each time we need a global synchronization, we call *sync* method on the object *grid*.

```
cooperative_groups::grid_group grid
 = cooperative_groups::this_grid();
grid.sync();
```

With regard to the previous VM [7] code rearrangement, we move a loop over SN steps inside a single kernel *run_SN* and insert the code of all previous VM kernels separated by the global synchronization calls. We run this kernel on the maximal grid structure among the previous VM kernels that corresponds to a rectangle of size $mm \times n$ initially applied to calculate the firing multiplicity on the incoming arcs of transitions. Note that other stages of the SN step employ this computing structure only partially. Though switching to a single kernel with a loop on SN steps inside it, speeds up the VM drastically by dozens of times as it is shown in Section 7.

Using the cooperative groups requires also a special implementation of the kernel call. Before the call, we specify an array *kernelArgs* of pointers to the parameters we are going to pass to the kernel. Then, we specify the block and the grid. In our VM, we use 1D grid of 1D blocks. And finally, we launch the kernel via function *cudaLaunchCooperativeKernel*, to which we pass the kernel address (*run_sn*), the grid and block specifications, and the list of kernel arguments:

```
void* kernelArgs[] = { (void*)&m,(void*)& n,(void*)& mm,(void*)& d_bi,
(void*)& d_bv,(void*)& d_di,(void*)& d_dv,(void*)& d_mu,(void*)& d_y,
(void*)&amp; d_f,(void*)&amp; maxk,(void*)& dbg };
dim3 block (mm);
dim3 grid  (n);
cudaLaunchCooperativeKernel((void*)run_sn,grid, block, kernelArgs);
```

Modified in the described way, VM [7] runs a few times faster which makes further amendments we developed look rather unessential, though they add-on to the VM's total enhanced performance as well. As the benchmarks acknowledge, the cooperative group application is limited by the grid size corresponding to the actual number of GPU streaming multiprocessors and their capacity. Thus, all the speed-up techniques described in this paper become significant for gaining additional speed-up, especially on big SN programs.

6. Composing SN machine for multicore CPU

Having an alternative, multicore CPU implementation of SN VM, brings in diversity and, taking into consideration the growing number of cores of modern CPU, for instance 288 for Monaka processor of Fujitsu [41], and the considerably higher performance of a CPU thread over a GPU thread (up to 10 times), sometimes, it provides better performance as benchmarks show, especially taking into consideration an overhead required for copying data between CPU and GPU.

The first CPU SN VM implementation [6] uses a conventional matrix representation of SN that limits its performance and processed data size. Recent implementations of SN VM for microcontrollers [3] with column-wise sparse matrices show rather good speed-up for runs on a single thread (core). In the present paper, preserving a uniform SN format, in the form of MCC, for both CPU and GPU, we enhance CPU SN VM performance by applying OpenMP [44] and the reduction of minimum and sequential choice. For nonstandard reduction of sequential choice, we develop ad-hoc functions.

Using the general mass-parallel chart of the VM step (Figure 3), we obtained the conventional code replacing a vector by a single loop and a cube by a nested loop, according to the chart design description [36]. SN VM CPU step, shown in Figure 3, is implemented by the three following code snippets. The first code snippet represents merged stages I and II to compute the vector *y* of transition firing multiplicities:

```
#pragma omp parallel for private (t,af,pi)
for(t=0; t<n; t++) {
       af=arc_firing(0,t);
       #pragma omp simd reduction(min:af)
       #pragma unroll
```

```
        for(pi=1; pi<mm; pi++) {
            af=zmin(af,arc_firing(pi,t));
        }
        y[t]=af;
}
```

In the above snippet, we also illustrate our work with MCC format using separate matrices to store indexes (prefix '_i ') and values (prefix '_v ') when computing the next marking. Computing the arc fireable condition is specified by macros *arc_firing*. Merging stages I and II of the SN VM step chart (Figure 3), allowed us to avoid using an auxiliary matrix; instead, an auxiliary vector *y* is in use. Of two nested loops, we start the outer loop in *parallel* mode of OpenMP while using *simd* facilities and *reduction* of minimum over the *unrolled* inner loop that results in better performance.

The sequential choice reduction can be implemented for a multicore CPU using OpenMP facilities based on the algorithm described in Section 3; also it can be obtained from the GPU code described in Section 5.2 via inserting the required number of loops with respect to the GPU grid structure as specified in [36]. Though the following code snipped of stage III is designed in a different way; it presents a technique of transforming sequential choice into computation of minimum that allows us to apply standard OpenMP reduction of minimum.

```
ipos_min=MU_MAX;
#pragma omp parallel for private (t,ipos) reduction(min:ipos_min)
for(t=0; t<n /* & ipos_min==MU_MAX*/ ; t++) {
        ipos=(y[t]>0)?t:MU_MAX;
        ipos_min=zmin(ipos_min,ipos);
}
```

We introduce a variable that stores the minimal index of the positive element of array *ipos_min* for the currently processed part of the array *y* and recompute the index of a 'current positive element' *ipos* that equals to infinity, represented with *MU_MAX* value, in case the current element equals zero. Thus, the loop computes the index of the first nonzero element. To optimize performance, we provide an extra condition for premature exiting the loop on encountering the first nonzero element that is prohibited in the reduction mode.

When there is a fireable transition indicated by *ipos_min* value lesser than *MU_MAX*, the stage IV for the marking recalculation, represented with the following code snippet, is run:

```
(f->c)=y[ipos_min]; (f->t)=ipos_min;
#pragma omp parallel for private (pi)
for(pi=0; pi<mm; pi++) { // next_mu
     if(bv[pi][f->t]>0) mu[ bi[pi][f->t] ] -= (f->c)*bv[pi][f->t];
     if(dv[pi][f->t]>0) mu[ di[pi][f->t] ] += (f->c)*dv[pi][f->t];
}
```

We suppose that the current step number is counted by the element $f->k$ of structure *f* that specifies the current state of the fireable transition choice process having such other elements as $f->t$ – for the firing transition number and $f->c$ – for the firing transition multiplicity; since the transition index can equal zero, we use the transition firing multiplicity to check whether there is a fireable transition at a step, otherwise, we break the basic loop over steps.

Let us consider the basic macros of SN VM. For the scalability of our virtual machines, both for GPU and CPU, we employ a conditional compilation to choose between *int* and *long*, based on the control symbol *_LONG_MU_*, for storing a place marking and a transition firing multiplicity:

```
//#define _LONG_MU_
#ifdef _LONG_MU_
  #define MUTY long
  #define MU_MAX LONG_MAX

#else
  #define MUTY int
  #define MU_MAX INT_MAX
#endif
#define arc_firing(pi,t) ((bv[(pi)][(t)]>0)?
   mu[ bi[(pi)][(t)] ] / bv[(pi)][(t)] : (bv[(pi)][(t)]<0)?
   ((mu[ bi[(pi)][(t)] ]>0)? 0: MU_MAX): MU_MAX)
```

The precompiler's conditional choice defines type *MUTY* and a constant *MU_MAX* for the VM code. To switch from default *int* to *long*, we just uncomment the definition of symbol *_LONG_MU_*. Macros *arc_firing(pi,t)* computes the current arc (from place indexed by variable *pi* to transition *t*) multiplicity, processing both regular and inhibitor arcs. Note that, we employ additional macros for work with dynamically allocated matrices; thus, the direct matrix indexation was used within simplified snippets only.

7. Comparative analysis of benchmarks

The developed SN VMs and utility programs, for instance for converting an SN from LSN/HSN format [6] to MCC format, accepted by the VMs of [7] and VMs described in this paper, have been uploaded for public use on GitHub [45] together with benchmark nets, that allows a reader to reproduce the benchmarks. The described in this section benchmarks have been obtained on HPC cluster Kelvin 2 [46]; basic dependencies are also confirmed by VMs' runs on the laptops and desktops of the authors. Conditional compilation features allow us to adjust memory allocated to store a place marking – integer or long integer type; we prefer signed types to avoid ongoing checks of the overflow.

7.1. Benchmark hardware and SN programs

We have been using HPC cluster Kelvin 2 [46] which brief description follows:

- 102 × 128 core Dell PowerEdge R6525 compute nodes with AMD EPYC 7702 dual 64-Core Processors (786 GB RAM).
- 8 High memory nodes (2TB RAM).
- 32 × NVIDIA Tesla v100 GPUs in 8 nodes.
- 16 × NVIDIA Tesla A100 GPUs in 4 nodes.
- 4 × AMD MI300X GPU's in 1 node.
- 4 × Nvidia H100 GPUs in 1 node.
- 4 × Intel Max 1100 GPUs in 1 node.
- 2PB of lustre parallel file system for scratch storage.

All compute nodes and storage are connected by EDR Infiniband fabric. Compute nodes run ROCKY8.10 operating system.

For benchmarks, we were using SN programs well described in [3, 6, 7], and briefly specified in Table 1.

In this paper, we focus on *de* and *mmul* benchmarks, which yield computations with rather rapidly growing computational complexity with respect to the parameter, using other benchmarks for complementary runs on desktops and laptops. We are taking into consideration these personal computing

Table 1. SN programs for benchmarks.

Net	Description	Parameter
de	Compute double exponent after Lipton	Power of double exponent
sadd	Sequence of additions	Number of additions
smul	Sequence of multiplications	Number of multiplications
poly	Compute polynomial of one variable	Power of polynomial
mmul	Multiply two square matrices	Size of matrix

devices to provide conditions for the wide spreading of SNC. We have *de* as a 'bad' and *mmul* as a 'good' benchmark from the point of view of mass parallel execution on GPU and multicore CPU.

7.2. Analysis of GPU and CPU benchmarks

We represent the benchmarks obtained on GPU and CPU in graphical form in Figure 7. We use a bar diagram, avoiding a misleading line approximation, because of the discrete character of the benchmark size parameter represented with a natural number. Also, we avoid using absolute values of time because they grow rather fast to represent a few benchmarks in the same graph without using a logarithmic scale. Because of this reason, we express information in the times of speed-up. We have chosen the double exponent and matrix multiplication benchmarks because they are time-consuming to obtain results almost independent of the operational environment's current state.

In Figure 7(a), we show bars for only two parameters, because for $n < 3$, nets run too fast to consider them, and for $n > 4$, nets require too much time, beyond one hour. In Figure 7(d), the benchmarks, for the cooperative group-based single kernel with global synchronization, finish for the matrix size 3, while the multiple-kernel implementation demonstrates perfect scalability, though its performance is considerably lower than the cooperative group's kernel.

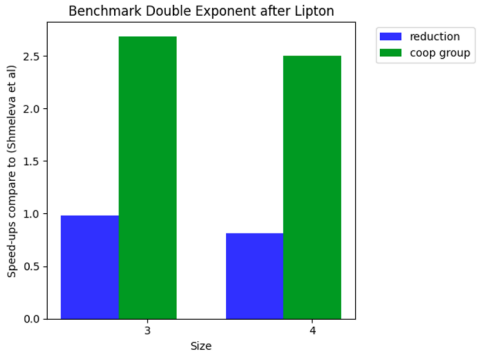
Benchmarks for multicore CPU, shown in Figure 7(c) and (d), demonstrate the rather good performance of the VMs on modern CPUs which can compete with GPU implementations of SN VM. Lipton's sequential, with multiple recursive returns, double exponent net *de* makes use of a single CPU thread's great performance to run dozens of times faster as shown in Figure 7(d) and (e). The mass parallel benchmark for matrix multiplication with the inline implementation of arithmetic operations *mmul* shows that with the growing size of the net, application of GPU is more advantageous illustrated with Figure 7(d) and (f).

7.3. Discussion of benchmarks

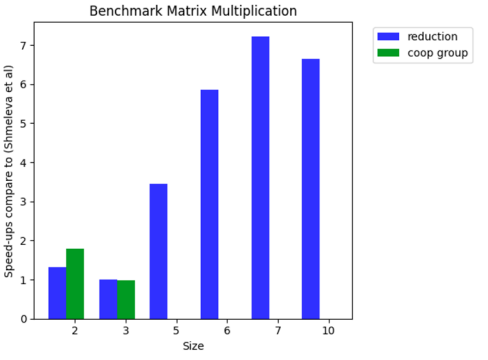
Using the cooperative groups feature of NVIDIA GPU architecture 90 increases considerably the number of threads running within a few blocks to synchronize, though the approach is limited by the actual number of GPU streaming multiprocessors and other GPU resources. Because of the limitation of the number of blocks for global synchronization, a good deal of speed-up, about 3 times, because of having the main loop over SN steps within a single kernel is possible to achieve for a rather modest number of blocks, actually reducing the maximal number of SN transitions to a few thousand. Thus, the only option to run big nets is to use a few kernels to implement an SN step and run them within a loop over steps implemented on the CPU.

The reduction of minimum gives an additional speed-up of about 20–30% for both single and multiple kernel implementations. An additional speed-up of about 20–40% is achieved for the multiple kernel implementations of the sequential choice reduction with a few levels of reduction over a block of maximal size. It is rather difficult to gain 10% additional speed-up for a single kernel implementation.

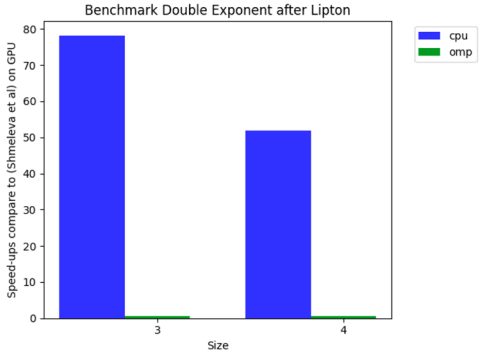
We observe also other aspects of the basic alternative using either a multiple kernel implementation compatible with NVIDIA GPU architecture 35 or a single core based on the cooperative groups features, in particular, the global synchronization of threads over a group of blocks or entire grid. Having multiple kernels allows us to adjust flexibly the grid specification for each kernel. For a single kernel,



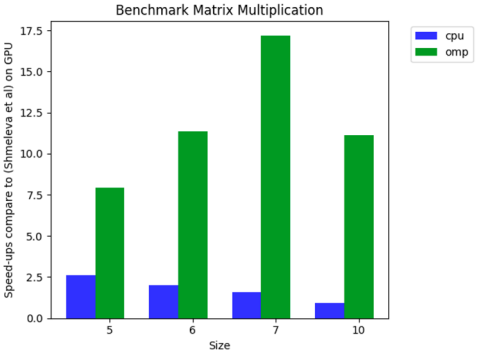
(a) double exponent on GPU;



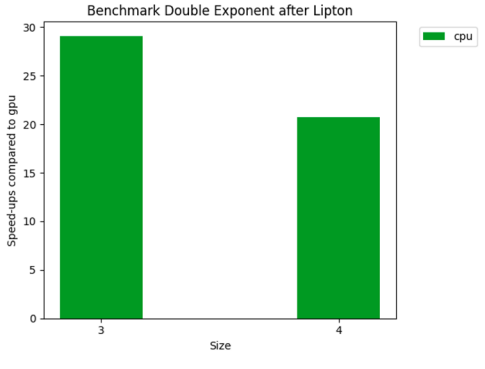
(b) matrix multiplication on GPU.



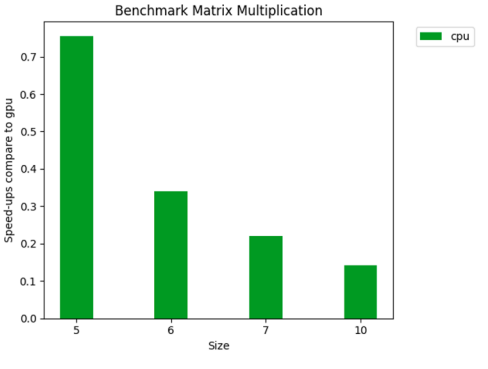
(c) double exponent on CPU;



(d) matrix multiplication on CPU.



(e) double exponent on CPU vs GPU;



(f) matrix multiplication on CPU vs GPU.

Figure 7. SN VM benchmarks. (a) double exponent on GPU. (b) matrix multiplication on GPU. (c) double exponent on CPU, (d) matrix multiplication on CPU. (e) double exponent on CPU vs GPU, (f) matrix multiplication on CPU vs GPU.

we should reach a certain compromise with regard to the grid specification more or less suitable for each stage of the SN step implementation.

Also, we would like to mention the fact that NVIDIA's rather aggressive marketing policy of stopping support of certain models of GPU after 5 years since their appearance on the market, makes a mass customer buying new models that is not well justified. In our case, we observed a rather good speed-up using the global synchronization though its application is restricted by the actual number of GPU SM to SN benchmark programs of a rather modest size. Benchmarks obtained on a Dell Inspiron 15,

5000 series laptop of 2015 with NVIDIA GPU RTX 640 for big SN programs (multiple kernel SN VM) are just about 2 times slower compared to benchmarks obtained on GPU Tesla H100 of Kelvin 2 UK HPC tier 2 cluster. To launch a CUDA program on RTX 640, we needed to run Linux Ubuntu 18.04 and gcc 5.2 to install CUDA 9.1 which supports our device.

Modern multicore CPUs, having more than a hundred cores, are competing with GPUs for the best performance when simulating discrete-event systems because of DES's sequential nature. Though, when implementing an SN step, we can compute the firing multiplicities on all incoming arcs of transitions, the following operation of minimum on each transition's incoming arcs, and, especially, of the sequential choice of firing transition, diminish the benefits. Even accelerated with reduction, they represent a bottleneck of VM implementations.

8. Conclusions

In the present paper, we studied the possibilities of SN VM for GPU speed-up using GPU novel architecture features, in particular, cooperative groups, and also the application of the reduction technique. A specific kind of reduction for sequential choice has been introduced and studied which has definite prospects of application for discrete-event system implementation on GPU. Speed-up of about 1.5–2 times has been obtained.

We found that the application of global synchronization facilities over GPU blocks, provided by the cooperative groups features of GPU architecture 90, yields 2–3 times speed-up using a single kernel with an internal loop on SN steps, though its application is limited by the maximal number of GPU blocks which may participate the global synchronization operations to run rather modest SN programs with the number of transitions of a few thousand.

Acknowledgments

We are grateful for use of the computing resources from the Northern Ireland High Performance Computing (NI-HPC) service funded by EPSRC (EP/T022175).

Disclosure statement

No potential conflict of interest was reported by the author(s).

ORCID

D. A. Zaitsev http://orcid.org/0000-0001-5698-7324
Y. Ajima http://orcid.org/0000-0003-0472-7453
J. F. C. Bartlett http://orcid.org/0009-0006-0551-5048
A. Kumar http://orcid.org/0009-0002-1408-6397

References

[1] Zaitsev DA. Sleptsov nets run fast. IEEE Trans Syst Man Cybern Syst. 2016;46(5):682–693. doi: 10.1109/TSMC.2015.2444414
[2] Zaitsev D. Sleptsov net computing resolves problems of modern supercomputing revealed by Jack Dongarra in his Turing award talk in November 2022. Int J Parallel Emerg Distrib Syst. 2023;38(04):275–279. doi: 10.1080/17445760.2023.2201002
[3] Xu R, Zhang S, Liu D, et al. Sleptsov net based reliable embedded system design on microcontrollers and FPGAs. In: 2024 IEEE International Conference on Embedded Software and Systems (ICESS). Wuhan, China: IEEE; 2024 Dec 13–15. doi: 10.1109/ICESS64277.2024.00011
[4] Vahidipour SM, Esnaashari M, Rezvanian A, et al. GAPN-LA: A framework for solving graph problems using petri nets and learning automata. Eng Appl Artif Intell. 2019;77:255–267. doi: 10.1016/j.engappai.2018.10.013
[5] Zaitsev DA. Universal Sleptsov net. Int J Comput Math. 2017;94(12):2396–2408. doi: 10.1080/00207160.2017.1283410
[6] Zaitsev DA, Shmeleva TR, Zhang Q, et al. Virtual machine and integrated developer environment for Sleptsov net computing. Parallel Process Lett. 2023;33(03):2350006. doi: 10.1142/S0129626423500068

[7] Shmeleva TR, Zaitsev ID, Retschitzegger W. GPU based virtual machine for Sleptsov net computing. Parallel Process Lett. 2024;34(03-04):2450012. doi: 10.1142/S0129626424500129
[8] Banks J, Carson JS, Nelson BL, et al. Discrete-event system simulation. 4th ed. Englewood Cliffs (NJ): Prentice Hall; 2005.
[9] Fujimoto RM. Parallel and distribution simulation systems. New York: Wiley-Interscience; 2000.
[10] Park H, Fishwick PA. A GPU-based application framework supporting fast discrete-event simulation. Simulation. 2010;86(10):613–628. doi: 10.1177/0037549709340781
[11] Perumalla KS. Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). In: PADS '06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation. IEEE Computer Society. Washington (DC): IEEE; 2006. p. 74–81.
[12] Faheem M, Murphy A, Reaño C. GPU-accelerated discrete event simulations: towards industry 4.0 manufacturing. In: 2021 IEEE Symposium on Computers and Communications (ISCC). Athens, Greece: IEEE; 2021. p. 1–7.
[13] Tang W, Yao Y. A GPU-based discrete event simulation kernel. Simulation. 2013;89(11):1335–1354. doi: 10.1177/0037549713508839
[14] Kunz G, Schemmel D, Gross J, et al. Multi-level parallelism for time- and cost-efficient parallel discrete event simulation on GPUs. In: 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation. Zhangjiajie, China; 2012. p. 23–32.
[15] Yianni PC, Neves LC, Rama D, et al. Accelerating petri-net simulations using NVIDIA graphics processing units. Eur J Oper Res. 2018;265(1):361–371. doi: 10.1016/j.ejor.2017.06.068
[16] Lagartinho-Oliveira C, Moutinho F, Gomes L. Reachability graph of IOPT petri net models using CUDA C++ parallel application. In: 11th Doctoral Conference on Computing, Electrical and Industrial Systems (DoCEIS). Costa de Caparica, Portugal; 2020. p. 93–100.
[17] Chalkidis G, Nagasaki M, Miyano S. High performance hybrid functional petri net simulations of biological pathway models on CUDA. IEEE/ACM Trans Comput Biol Bioinform. 2011;8(6):1545–1556. doi: 10.1109/TCBB.2010.118
[18] Bunch R, Rose DJ. Sparse matrix computations. London: Academic Press; 1976.
[19] Davis TA. Direct methods for sparse linear systems. Philadelphia: SIAM; 2006.
[20] Groote JF, Mousavi MR. Modeling and analysis of communicating systems. Cambridge: The MIT press; 2014.
[21] Cervesato I, Durgin N, Mitchell J, et al. Relating strands and multiset rewriting for security protocol analysis. In: Proceedings 13th IEEE Computer Security Foundations Workshop, CSFW-13. Cambridge, UK: IEEE; 2000. p. 35–51.
[22] Rosa-Velardo F. Multiset rewriting: a semantic framework for concurrency with name binding. In: Ölveczky, PC, editor. Rewriting logic and its applications. WRLA 2010. Berlin: Springer; 2010. (Lecture notes in computer science; vol. 6381).
[23] Yuan C. Place/Transition Net Systems. In: Principle of Petri Nets. Singapore: Springer; 2025.
[24] Johnson R. GPU assembly and shader programming for compute: low-level optimization techniques for high-Performance parallel processing. University of Warwick, Warwick, UK: HiTeX Press; 2025.
[25] Cheng J, Grossman M. Professional CUDA C programming. Indianapolis (IN): Wiley; 2014.
[26] CUDA C++ Programming Guide. NVIDIA. Available from: https://docs.nvidia.com/cuda/cuda-c-programming-guide
[27] Martin PJ, Ayuso LF, Torres R, et al. Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In: 2012 International Conference on High Performance Computing & Simulation (HPCS). Madrid, Spain; 2012. p. 511–519.
[28] Jradi WAR, Nascimento HADdo, Santos Martins W. A fast and generic GPU-based parallel reduction implementation. In: 2018 Symposium on High Performance Computing Systems (WSCAD). Sao Paulo, Brazil; 2018. p. 16–22.
[29] Zaitsev DA. Solving operative management tasks of a discrete manufacture via Petri net models [PhD thesis]. Kiev: the Academy of Sciences of Ukraine, Institute of Cybernetics name of V.M. Glushkov; 1991. In Russ. Available from: https://daze.ho.ua/daze-phd-1991.pdf
[30] Zaitsev DA. Strong Sleptsov nets are turing complete. Inf Sci. 2023;621:172–182. doi: 10.1016/j.ins.2022.11.098
[31] Berthomieu B, Zaitsev DA. Sleptsov nets are turing-complete. Theor Comput Sci. 2024;986:114346, doi: 10.1016/j.tcs.2023.114346ISSN 0304–3975.
[32] Petri CA. Kommunication mit automaten [PhD thesis]. Darmstadt: Technischen Hoschule Darmstadt; 1962.
[33] Murata T. Petri nets: properties, analysis and applications. Proc IEEE. 1989;77(4):541580. doi: 10.1109/5.24143
[34] Zaitsev DA, Jurjens J. Programming in the Sleptsov net language for systems control. Adv Mech Eng. 2016;8(4):1–11. doi: 10.1177/1687814016640159
[35] Berthomieu B, Ribet P-O, Vernadat F. The tool TINA construction of abstract state spaces for petri nets and time petri nets. Int J Production Res. 2004;42(14):2741–2756. doi: 10.1080/00207540412331312688
[36] Zaitsev DA, Zhang Z, Liu D, et al. Notation for mass parallel algorithms: computing petri net state space on GPU case study. Int J Parallel, Emergent Distrib Syst. 2025;40(2):101–115. doi: 10.1080/17445760.2024.2431545
[37] Graph 500: large-scale benchmarks. Available from: https://graph500.org
[38] Shimizu T. Supercomputer Fugaku: Co-designed with application developers/researchers. In: 2020 IEEE Asian Solid-State Circuits Conference (A-SSCC). Hiroshima, Japan: IEEE; 2020. p. 1–4.
[39] Ajima Y, et al. The Tofu interconnect D. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). Belfast, UK: IEEE; 2018. p. 646–654. doi: 10.1109/CLUSTER.2018.00090

[40] Sinjo N. From Past to Future : The Legacy and Hypothesis of Supercomputing, ICS 2024. Available from: https://ics2024.github.io/slides/20240606_ICS2024.pdf
[41] Fujitsu. next arm processor FUJITSU-MONAKA and its technologies. Available from: https://www.fujitsu.com/global/imagesgig5/FUJITSU-MONAKA.pdf
[42] Dalmia P, Mahapatra R, Intan J, et al. Improving the scalability of GPU synchronization primitives. IEEE Trans Parallel Distrib Syst. 2023;34(1):275–290. doi: 10.1109/TPDS.2022.3218508
[43] Yang C, Buluç A, Owens JD. Design principles for sparse matrix multiplication on the GPU. In: Aldinucci, M, Padovani, L, Torquati, M, editors. Euro-Par 2018: Parallel Processing. Euro-Par 2018. p. 672–687. (Lecture Notes in Computer Science; vol. 11014).
[44] Chapman B, Jost G, van der Pas R. Using openMP: portable shared memory parallel programming. Cambridge (MA): MIT Press; 2007.
[45] Zaitsev D. Tools for Sleptsov net computing. 2025. Available from: https://github.com/dimazaitsev/SNCtools
[46] Kelvin2. Northern Ireland high performance computing. Available from: https://www.ni-hpc.ac.uk/Kelvin2