Heterogeneous Combinatorial Candidate Generation

Fahad Khalid[1], Zoran Nikoloski[2], Peter Tröger[1], and Andreas Polze[1]

[1] Hasso Plattner Institute for Software Systems Engineering
{fahad.khalid,peter.troeger,andreas.polze}@hpi.uni-potsdam.de
[2] Max Planck Insitute of Molecular Plant Physiology
nikoloski@mpimp-golm.mpg.de

Abstract. Elementary Flux Modes (EFMs) can be used to characterize functional cellular networks and have gained importance in systems biology. Enumeration of EFMs is a compute-intensive problem due to the combinatorial explosion in candidate generation. While there exist parallel implementations for shared-memory SMP and distributed memory architectures, tools supporting heterogeneous platforms have not yet been developed. Here we propose and evaluate a heterogeneous implementation of combinatorial candidate generation that employs GPUs as accelerators. It uses a 3-stage pipeline based method to manage arithmetic intensity. Our implementation results in a 6x speedup over the serial implementation, and a 1.8x speedup over a multithreaded implementation for CPU-only SMP architectures.

1 Introduction

Metabolism is the collection of chemical compounds, called metabolites, transformed via enzymatic reactions to sustain the functions of biochemical systems. The network structure of metabolism can be characterized by a directed weighted hypergraph [1] in which directed hyperedges represent reactions and nodes stand for metabolites. The number of molecules with which a metabolite participates as a substrate and/or product in a reaction specifies the reaction-specific stoichiometry of the metabolite, rendering the hypergraph node-weighted. The concept of a steady state, *i.e.*, equilibrium, whereby there is no change in concentrations of the considered metabolites, is often employed in analyzing the functional behavior of (large-scale) metabolic networks [2].

Interestingly, the steady-state behavior of metabolic networks, described only by the directed weighted hypergraph, can be fully characterized by the minimal subnetworks which operate at equilibrium, referred to as elementary flux modes [3, 4] (EFMs). Due to the minimality condition, an EFM cannot operate in a steady state upon removal of any of its components (*i.e.*, reactions or metabolites). Further, EFMs provide a mathematical definition for the concept of a biochemical pathway. Since EFMs can capture emergent functions of biochemical systems, they have been used to analyze key systemic properties, including robustness and flexibility [5]. However, characterization of a system's behavior by means of EFMs requires their enumeration, which involves systematic evaluation of all possible subnetworks with respect to several constraints/conditions they must satisfy. This process is combinatorial in nature and expensive in terms of both computational and memory requirements, thus, limiting application to systems of small size. Therefore, parallelization of the existing approaches for EFM enumeration is necessary for large-scale networks.

Both shared-memory and distributed-memory parallel approaches have been developed. A parallel out-of-core implementation [6] was one of the first attempts at parallelization. Another implementation, the *efmtool* [1] is targeted towards shared memory SMP architectures. It is based on the state-of-the-art in algorithmic approach for EFM enumeration [7–9], and has been used by scientists other than the developers to report important results [10].

The *ElMo-Comp* tool [11] was designed specifically for distributed memory architectures. Since the first public release, the tool has been extended to handle larger networks. The first extension [12] employs the *Divide and Conquer* strategy, where the complete set of EFMs is partitioned into disjoint subsets that can be processed independently. In the second extension [13], the concept of *Partitioned Global Address Space (PGAS)* is utilized to enable sharing of memory resources across the cluster.

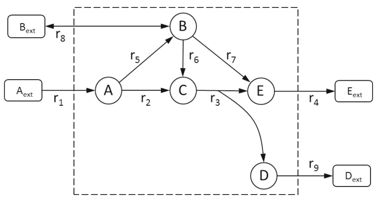
Our study builds on *ElMo-Comp* [11] and extends it to support heterogeneous architectures with GPUs as accelerators. Our efforts are focused on the computational bottleneck, which is the memory-bound part of the algorithm we term *combinatorial candidate generation.*

The paper is organized as follows: Section 1.1, presents the mathematical model and algorithm used for EFM enumeration, including the computational bottleneck. Our approach is presented in Section 2, followed by evaluation in Section 3. Related work is discussed in Section 4, followed by conclusion and future work.

1.1 Enumeration of Elementary Modes

Mathematical Model. Consider the hypergraph representation of a paradigmatic metabolic network presented in Figure 1A (adapted from [14]). Metabolites in the network are represented by nodes and reactions by hyperedges. Metabolites are divided into two groups *internal* to the system and *external*, *i.e.*, in the systems environment, delineated by the dotted line box in Figure 1A. The reactions that involve both external and internal metabolites are termed exchange reactions. Moreover, with respect to directionality, reactions are divided into reversible and irreversible, belonging to the sets *Rev* and *Irr*, respectively. In Figure 1A, reaction r_8 is reversible. The information encoded in the directed weighted hypergraph can be captured by the corresponding stoichiometric matrix. For a metabolic network with m metabolites and q reactions, the stoichiometric matrix, S, consists of m rows and q columns. The entry $S_{i,j}$ quantifies the number of molecules with which metabolite i participates in reaction j, and

[1] efmtool - Elementary Flux Mode Tool: `http://www.csb.ethz.ch/tools/efmtool`



$$
S = \begin{array}{c} \\ A \\ B \\ C \\ D \\ P \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} r1 & r2 & r3 & r4 & r5 & r6 & r7 & r8 & r9 \end{array} \\
\begin{pmatrix}
1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & -1 & -1 & 0 \\
0 & 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 1 & -1 & 0 & 0 & 2 & 0 & 0
\end{pmatrix}
\end{array}
$$

Fig. 1. (A) Metabolic network (B) corresponding stoichiometric matrix

the sign indicates if the metabolite participates as a substrate (negative) or a product (positive), illustrated in Figure 1B.

Reaction rates, called fluxes, quantify the behavior of reactions transforming the metabolites in the network. The steady state of a metabolic network specified by its stoichiometric matrix can be characterized in terms of a q-vector termed flux vector (or distribution), denoted by v. Reaction rates can be used to characterize the change in the concentration of metabolites, since $\frac{dX}{dt} = Sv$, where X is an m-vector gathering the concentrations of metabolites. In a steady state, there is no change in concentrations, and the steady-state flux distribution can be determined by solving the system of linear equations:

$$Sv = 0, \tag{1}$$

whereby v belongs to the nullspace of the stoichiometric matrix S. Since the number of metabolites is usually smaller than the number of reactions, the system in Eq. (1) is underdetermined and usually results in an infinite number of solutions. Note that the system of linear equations is homogeneous if all metabolites are internal; otherwise, the system is inhomogeneous. Moreover, a steady-state flux distribution is further constrained by the reaction directionalities, so that fluxes of irreversible reactions must be non-negative, *i.e.*,

$$v_i \geq 0, \forall i \in Irr \tag{2}$$

By combining the steady-state and directionality constraints imposed by Eq.(1) and Eq.(2), respectively, the solution space forms a convex polyhedral cone, P, defined as [15]:

$$P = \{v \in \mathbb{R}^q \mid Sv = 0, v_i \geq 0, \forall i \in Irr\}, \tag{3}$$

where $\mathbb{R}^q$ is the q-dimensional vector space in real numbers. Clearly, every vector that lies in the cone represents a feasible flux distribution in the metabolic network. With this notation, we need the elementarity constraint to define an EFM. Let $supp(v) = \{i \mid v_i \neq 0\}$ and let E denote the set of all EFMs, then, the following holds:

$$\forall v \in E, \nexists x \in E \mid v \neq x, supp(v) \subseteq supp(x). \tag{4}$$

If the system consists only of irreversible reactions, the solution space forms a pointed polyhedral cone [5]. In this case, the set of elementary modes comprises a unique minimal set of generating vectors for the entire flux space.

The Nullspace Algorithm. All algorithms for elementary mode enumeration are based on the Double Description Method [16] for extreme ray enumeration of a polyhedral cone, well-studied in computational geometry. These algorithms vary primarily in the order in which the steady-state and reaction reversibility constraints are processed. Our study is based on the Nullspace algorithm [17] (see [18] for detailed description). Here, we present a brief sketch, highlighting only the most relevant steps. The Nullspace algorithm is summarized as follows:

1. The stoichiometric matrix S is compressed using methods specified in [5]. Let the compressed matrix be $S'_{m \times q}$. Then the nullspace obtained by solving $S'v = 0$ is denoted by K'. Each row in K' corresponds to a reaction, and each column represents a potential EFM. Let I denote the identity matrix. The compressed nullspace is permuted to obtain the following form:

$$K' = \begin{pmatrix} R^{(1)} \\ R^{(2)} \end{pmatrix} = \begin{pmatrix} I \\ R^{(2)} \end{pmatrix} \tag{5}$$

   where the directionality constraints are already solved for rows in I. Directionality constraints must now be applied to all rows in $R^{(2)}$.
2. For each row in $R^{(2)}$:
   (a) Generate bitwise combinations of selected columns in $R^{(1)}$ to produce candidate bit vectors. Given a threshold τ, a candidate vector, v, $supp(v) > \tau$ is discarded,
   (b) Remove duplicate candidate vectors,
   (c) Verify each candidate for elementarity,
   (d) Generate algebraic combinations on the current row in $R^{(2)}$,
   (e) Convert the current row in $R^{(2)}$ to the corresponding binary representation and move it to $R^{(1)}$,
   (f) Append the generated EFMs as column vectors to the nullspace.

We note that once a row in $R^{(2)}$ is processed, it can be converted to a binary representation and moved to $R^{(1)}$ [5]. Moreover, in *ElMo-Comp*, $R^{(1)}$ is compressed by a factor equal to the machine word length, *i.e.*, 32 or 64 times. Finally, once all reactions have been processed, columns of the kernel matrix represent all EFMs for the given network.

Combinatorial Candidate Generation. The most compute intensive step in the Nullspace algorithm is the generation of combinations in $R^{(1)}$ (see Algorithm 1). We refer to this step as *combinatorial candidate generation*. This step being the computational bottleneck is the primary focus of our work.

Algorithm 1 consists of two core computational operations: a bitwise OR between two columns that results in a candidate vector (Line 3) and a *popcount* on the candidate vector (Line 4). To process a single candidate, we need *two* arithmetic and *four* memory access operations (assuming *popcount()* is available as a hardware instruction). Moreover, two read operations are required to fetch the input columns, and two write operations are required to store the indices corresponding to the input columns. The data type used for both input and output values is 64-bit unsigned integer. As compared to the 32-bit data types, this increases the size of the input and output values, halves the throughput of the two operations, and, thus, results in a very low compute-to-memory-access ratio. Therefore, combinatorial candidate generation can be classified as a memory-bound algorithm with low arithmetic intensity.

Algorithm 1: Serial combinatorial candidate generation. Index vectors contain column indices of the corresponding matrices; OR is the binary bitwise OR operation; τ is a threshold (as described in Section 1.1)

```
Input  : Bit matrices: MatrixA, MatrixB
         Index vectors: IndicesA, IndicesB
         Integer: $\tau$
Output: Candidate column index pairs of the form
        $\{(a, b) \mid a \in IndicesA \text{ and } b \in IndicesB\}$
foreach colA: column in MatrixA do
    foreach colB: column in MatrixB do
        candidate = MatrixA[colA] OR MatrixB[colA];
        nonZeros = popcount(candidate);
        if nonZeros $\leq \tau$ then
            store index pair (IndicesA[colA], IndicesB[colB]);
        end
    end
end
```

In massively parallel accelerator architectures, like GPUs, most of the chip area is dedicated to arithmetic and logic units, which results in very small sizes for fast on-chip memory. Therefore, these architectures are ill-suited for algorithms with low arithmetic intensity. Here, we present an approach that renders it possible to exploit the massive parallelism of GPUs, leading to significant speedup despite the memory-bound nature of the combinatorial candidate generation algorithm.

2 Our Approach

In the rest of the document, we assume all index values to be of type 64-bit unsigned integer. We refer to the CPU as *Host*, and the GPU as *Device*.

Algorithm 1 can be decomposed into two parts: (1) generation of a candidate vector followed by popcount (Lines 3,4), which results in a Boolean; and (2) storage of input column indices (Line 6). These two parts can be implemented as distinct phases — *Generate* and *Map*.

Generate Phase. This phase is implemented as the *Device* kernel (see Algorithm 2). As in the serial version, the two input values are fetched to perform the bitwise OR (Line 3) and popcount (Line 4) operations. Since the max-non-zero condition is the final step in this phase, instead of storing two 64-bit integers, the kernel stores a single Boolean value (Line 5). This significantly increases the arithmetic intensity, and hence the kernel performance. Nevertheless, storing a Boolean value against each possible combination has a disadvantage. For two matrices A and B of sizes m and n, respectively, the size of the output Boolean array is $m \times n$. As the number of reactions in the network increases, the size of the output array becomes too large for the *Device* memory. This results in large and frequent *Device*-to-*Host* memory transfers that become a serious bottleneck.

Algorithm 2: GPU kernel for combinatorial candidate generation

```
  Input  : Matrix A, Matrix B, $\tau$
  Output: Result - bit array
1 for 1 to compressionFactor do
2     compute indexMatA, indexMatB, indexResult ;    // index algebra
3     candidate = MatrixA[indexMatA] OR MatrixB[indexMatB];
4     nonZeros = popcount(candidate);
5     result[indexResult] = (nonZeros $\leq \tau$);
6 end
```

We address this problem by introducing a *compression factor*, which defines the number of result values generated by each *Device* thread. Instead of storing the output in a Boolean variable, we use a single bit. Therefore, a single thread can generate and store 64 values in a single 64-bit unsigned integer. This reduces the size of the output array by *compression factor*, and *Device*-to-*Host* transfers no longer constitute the bottleneck. The data type of the output array is independent of the other data types used, and its size merely indicates the maximum number of results generated by a single thread.

Map Phase. Once kernel execution is complete, the index of each bit in the result array corresponds to a candidate, and the bit value indicates whether the candidate should be considered for further processing. The index of each set bit must then be mapped to the corresponding index pair used to generate the corresponding value (a candidate is identified by the corresponding pair of input columns). The *Map* phase traverses the output bit array, looks for set bits, and maps their indices to the corresponding input index pairs. This phase is highly memory-bound, and thus performs better on the *Host*.

2.1 Concurrent *Host–Device* Processing

The *Generate-Map* strategy with compression factor, results in a significant speedup over both the serial version, and a naïve kernel without compression factor. In the subsections to follow, we show how *pipeline parallelism* [19] can be employed to implement concurrent *Device-Host* processing for larger speedup.

Two-Stage Pipeline: *Device* only Due to the combinatorial nature of the algorithm, a very large number of *Device* threads are required to compute all possible combinations. For NVIDIA GPUs with compute capability up to 2.0, the maximum number of thread blocks is limited to 65535. Therefore, for larger input sizes, multiple grid (batches of threads) executions are required. Between two subsequent grid executions, the result array has to be transferred from *Device* to *Host*, so that enough space is left on the *Device* to hold the result array for the next grid in line for execution. Moreover, the *Map* phase can only begin once the final *Device*-to-*Host* memory transfer is complete. This results in a serial processing of stages as depicted in Figure 2a.

The NVIDIA CUDA programming model provides API that makes it possible to overlap kernel execution and memory transfer. Using this feature, one kernel can be launched after the other without waiting for a *Device*-to-*Host* memory transfer operation to finish. This results in a 2-stage pipeline as depicted in Figure 2b. To ensure that a memory transfer only begins after the corresponding kernel computation is finished, an event notification system is employed. Each kernel executes in its own stream [20], and once the kernel execution is complete, an event notification is sent from the kernel stream to the corresponding memory transfer stream. The memory transfer stream begins operation only after the event notification has been received.

Three-Stage Pipeline: *Device* and *Host* The CUDA stream based event notification system has traditionally been limited to event exchange among *Device* operations only. With the release of CUDA 5.0 however, it is now possible to register *Host* callback functions with *Device* streams. These callbacks make it possible for the *Device* to send stream event notifications to the *Host*. We utilize the callback feature to extend the 2-stage pipeline. The *Map* phase is included as an additional stage, resulting in a 3-stage pipeline that spans both *Host* and *Device* functions. The concept is illustrated in Figure 2c.

The process starts by calculating the total size of the result array, and splitting it into multiple segments. For each segment, one or more kernels are launched in different streams, which we refer to here as grid streams. Events are recorded for each grid stream, on which the asynchronous memory transfer operation waits. Once all grids for the current segment have finished execution, the *Device*-to-*Host* memory transfer begins. At the same time, grids are launched for the next segment. Once the *Device*-to-*Host* memory transfer operation is complete, it

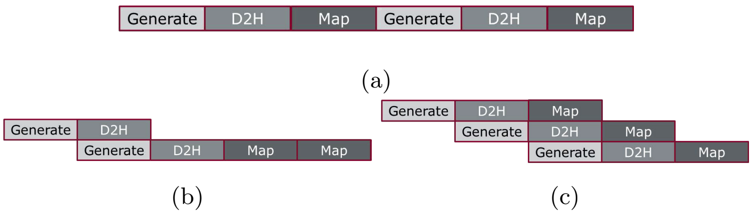


Fig. 2. Illustration of (a) serial, (b) 2-stage, (c) 3-stage processing of phases. D2H is *Device*-to-*Host* memory transfer

calls a *Host* function registered as callback with the memory transfer stream. The callback function sets a flag, indicating that the memory transfer operation for a specific segment is complete. The flag serves as a notification for the *Map* phase, and triggers the map operation on the segments result. To ensure that the *Generate* and *Map* phases execute not just concurrently but in parallel, these operations are executed by two separate *Host* threads that share data structures for the exchange of event notifications.

The memory size on the *Device* is generally smaller than that of the *Host*. Therefore, it is the programmers responsibility to ensure that all datasets that fit into the *Host* memory can also run on the *Device*. For this purpose, we use the concept of *partition*. Maximum partition size depends on the total amount of global memory available on the *Device* and the sizes of the input and result data structures. Large datasets are split into multiple partitions, where each partition consists of multiple segments. The *Device* utilizes the 3-stage pipeline to process one partition at a time. The next partition can start execution only if the required memory resources have been released by the previous partition.

2.2 Overall Architecture

In order to utilize all available processing resources, the candidate generation algorithm is processed using two implementations that execute in parallel (as shown in Figure 3). One is a multithreaded OpenMP based (*Host*-only) implementation that executes solely on the CPU cores. The other is the 3-stage *Device-Host* pipeline as describer in Section 2.1.

A *Host* thread decomposes the input data structures into two parts for distribution amongst the *Host*-only implementation and the 3-stage pipeline. These are passed on to a *parallel harness* that invokes two threads. The first thread invokes the OpenMP based *Host*-only multithreaded implementation. The second invokes the *device harness*. The device harness defines the data structures to be shared amongst the *Generate* and *Map* phases, and instantiates these phases as two OpenMP threads. The *Generate* thread further manages the massively parallel execution on the *Device*. The *Map* thread listens to memory transfer completion events and invokes the bit-to-input-index mapping routine. This routine is also implemented as multithreaded OpenMP code, which helps speedup the mapping process. Once both the *Host*-only code and device harness threads have completed execution, a reduction operation is performed to consolidate the results.

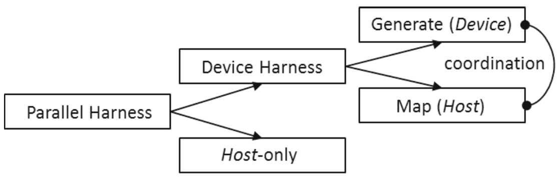


Fig. 3. Thread hierarchy. *Parallel harness* spawns *device harness* and *Host-only* code. *Device harness* spawns the *Generate* and *Map* phases.

3 Evaluation

We present comparative results from three different implementations: the serial Nullspace program available in *ElMo-Comp* [11]; our OpenMP based shared-memory parallel implementation for SMP architectures; and our 3-stage pipeline implementation for heterogeneous architectures. Both our implementations are based on the *ElMo-Comp* code base.

3.1 Test Environment

The machine used for running the reported experiments has 24GB of main memory, and consists of an Intel Xeon E5620 CPU and an NVIDIA Tesla 2050 GPU. The Xeon E5620 processor is based on the Nehalem EX architecture, supporting a 64-bit instruction set with SSE 4.2. It has 4 cores, each supporting 2 hardware threads. The Tesla 2050 GPU is based on the Fermi architecture, with compute capability rating of 2.0. The operating system used is Ubuntu SMP 12.04. The code was compiled using GCC 4.4.3 and NVCC with CUDA 5.0.

3.2 Results

Table 1 summarizes results of three implementations against five real networks of varying sizes, all capturing E.Coli central metabolism. The candidate generation step is performed on compressed networks [5]. Accordingly, network sizes mentioned in the table correspond to compressed networks.

The results show that the utility of the heterogeneous implementation increases with the number of candidate vectors generated during execution. Poor performance of the heterogeneous implementation against small networks is attributed to the overhead incurred by the transfer of data between *Host* and *Device* memory, as well as coordination between the *Generate* and *Map* phases. For larger networks, the incurred overhead is overshadowed by the performance gain. The *Host*-only multithreaded implementation utilizes all 8 threads available on the processor. The heterogeneous implementation dedicates 2 threads to the *Host*-only implementation, and 2 threads to the *Map* phase. Input data is distributed amongst the device harness and the *Host*-only implementation such that only $\frac{1}{8}th$ of the input size is processed by the *Host*-only code, while the rest is processed by the device harness.

It is important to properly tune compression factor and maximum segment size for optimal performance. A higher value of compression factor translates to more work per *Device* thread, and lower *Device* output size. Maximum segment size defines the maximum result array size for which a memory transfer to the *Host* must be initiated. Together these two parameters balance the speed and coordination between the pipeline stages. For the results presented in Table 1, compression factor is 64 and the maximum segment size is 60 MB.

Table 1. Comparative results of three different implementations against five networks. Execution times (in seconds) are presented against each implementation and network; m is the number of metabolites, q is the number of reactions.

Network Size	#candidates	Time (s)		
		Serial	OpenMP	Pipelined
26m × 38q	219743731	1.2	0.38	0.39
26m × 40q	130992739	0.77	0.35	0.35
26m × 41q	752482917	4.1	1.6	1.0
27m × 43q	2616975505	14	5.5	2.5
29m × 45q	122559991284	690	150	110

4 Discussion

Related Work. Recently, major vendors from the hardware and software industries have pointed out the significance of considering arithmetic intensity for decisions concerning suitable hardware. Empirical results [21] were presented to show how Floating Point Operations Per Second (FLOPS) is not an adequate measure for memory-bound algorithms. In addition, the case of Sparse Matrix-vector Multiplication (SpMV) was used to study accelerator (GPU) performance for algorithms with low arithmetic intensity [22]. The authors conclude that with the coming generations of processors, in comparison to GPUs, CPUs are becoming more and more suitable for such problems.

A multitude of scientific applications have been recently designed take advantage of heterogeneous architectures with GPUs as accelerators. These include linear solvers [23], solvers for path problems in graphs [24], as well as applications for simulation science [25], to name a few. Moreover, there have been efforts to increase the arithmetic intensity of certain algorithms [26], so that the processing resources can be utilized effectively.

Conclusions and Future Work. We presented a novel method to utilize GPUs for *combinatorial candidate generation*, a specific memory-bound algorithm, required for EFM enumeration. Our approach focuses on the concurrent, coordinated, *Device-Host* pipelined execution model. This approach is feasible due to the possibility to split the algorithm into two phases, where the phase of a high arithmetic intensity is executed on a GPU, while the other is executed concurrently on the *Host*. The 2-phase computation points to the Map-Reduce Pattern for Parallel Computation [27]. We conjecture that memory-bound algorithms amenable to this pattern may also benefit from the approach presented in this paper.

A large number of important combinatorial algorithms (such as those employed in network analysis on big data [28]) are memory-bound. In order to effectively utilize accelerators for such algorithms, novel methods for managing arithmetic intensity must be developed. The approach presented in this paper is a first step in this direction.

However, despite its effectiveness, the presented approach has certain drawbacks. Only limited functionality is available in the CUDA programming model to facilitate *Device-Host* pipelining; for instance, *Host* memory must be page-locked [20] (which is scarce), and merely a restricted set of operations is permissible within a callback. This complicates *Device-Host* coordination, and requires a greater number of parameters to be tuned for optimal performance.

In the future, we intend to tailor the application for execution on heterogeneous clusters. This requires support for multi-GPU execution. Also, we have only presented acceleration of one of the steps in the Nullspace algorithm. Work is underway to assess the feasibility of heterogeneous implementations for other steps. In addition to acceleration, efficient memory management is a vital factor for processing of large networks. The number of EFMs grows almost exponentially with the input size [29], which puts very high demands on memory. We are currently in the process of devising better compression techniques, as well as strategies for efficient data distribution on distributed memory architectures.

References

1. Klamt, S., Haus, U.U., Theis, F.: Hypergraphs and cellular networks. PLoS Comput. Biol. 5(5), e1000385 (2009)
2. Heinrich, R., Schuster, S.: The Regulation of Cellular Systems. Springer (1996)
3. Schuster, S., Fell, D.A., Dandekar, T.: A general definition of metabolic pathways useful for systematic organization and analysis of complex metabolic networks. Nat. Biotech. 18(3), 326–332
4. Papin, J.A., Price, N.D., Wiback, S.J., Fell, D.A., Palsson, B.O.: Metabolic pathways in the post-genome era. Trends Biochem. Sci. 28(5), 250–258 (2003)
5. Gagneur, J., Klamt, S.: Computation of elementary modes: a unifying framework and the new binary approach. BMC Bioinformatics 5(1), 175 (2004)
6. Samatova, N.F., Geist, A., Ostrouchov, G., Melechko, A.V.: Parallel out-of-core algorithm for genome-scale enumeration of metabolic systemic pathways. In: Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS 2002, p. 249. IEEE Computer Society, Washington, DC (2002)
7. Terzer, M., Stelling, J.: Accelerating the computation of elementary modes using pattern trees. In: Bücher, P., Moret, B.M.E. (eds.) WABI 2006. LNCS (LNBI), vol. 4175, pp. 333–343. Springer, Heidelberg (2006)
8. Terzer, M., Stelling, J.: Large-scale computation of elementary flux modes with bit pattern trees. Bioinformatics 24(19), 2229–2235 (2008)
9. Terzer, M., Stelling, J.: Parallel extreme ray and pathway computation. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part II. LNCS, vol. 6068, pp. 300–309. Springer, Heidelberg (2010)
10. Jungreuthmayer, C., Ruckerbauer, D.E., Zanghellini, J.: Utilizing gene regulatory information to speed up the calculation of elementary flux modes. arXiv:1208.1853 [q-bio.MN]
11. Jevremović, D., Trinh, C.T., Srienc, F., Sosa, C.P., Boley, D.: Parallelization of nullspace algorithm for the computation of metabolic pathways. Parallel Computing 37(6-7), 261–278 (2011)

12. Jevremović, D., Boley, D., Sosa, C.: Divide-and-conquer approach to the parallel computation of elementary flux modes in metabolic networks. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 502–511 (May 2011)
13. Jevremović, D., Boley, D.: Parallel computation of elementary flux modes in metabolic networks using global arrays. In: The 6th Conference on Partitioned Global Address Space Programming Models (2012)
14. Trinh, C.T., Wlaschin, A., Srienc, F.: Elementary mode analysis: a useful metabolic pathway analysis tool for characterizing cellular metabolism. Appl. Microbiol. Biotechnol. 81(5), 813–826 (2009)
15. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1998)
16. Fukuda, K., Prodon, A.: Double description method revisited. In: Deza, M., Euler, R., Manoussakis, I. (eds.) CCS 1995. LNCS, vol. 1120, pp. 91–111. Springer, Heidelberg (1996)
17. Wagner, C.: Nullspace approach to determine the elementary modes of chemical reaction systems. J. Phys. Chem. B 108(7), 2425–2431 (2004)
18. Jevremović, D., Trinh, C.T., Srienc, F., Boley, D.: On algebraic properties of extreme pathways in metabolic networks. J. Comput. Biol. 17(2), 107–119 (2010)
19. Dongarra, J., Foster, I., Fox, G.C., Gropp, W., Kennedy, K., Torczon, L., White, A. (eds.): The Sourcebook of Parallel Computing. Morgan Kaufmann (2002)
20. NVIDIA: CUDA C programming guide. Design Guide PG-02829-001_v5.0 (October 2012)
21. Mora, J.: Do theoretical flops matter for real application performance? In: HPC Advisory Council Spain Workshop (2012)
22. Davis, J.D., Chung, E.S.: Spmv: A memory-bound application on the gpu stuck between a rock and a hard place. Technical report, Microsoft Research Silicon Valley (September 2012)
23. Kurzak, J., Luszczek, P., Faverge, M., Dongarra, J.: LU factorization with partial pivoting for a multicore system with accelerators. IEEE Transactions on Parallel and Distributed Systems PP(99), 1 (2012)
24. Buluç, A., Gilbert, J.R., Budak, C.: Solving path problems on the GPU. Parallel Computing 36(5-6), 241–253 (2010)
25. Domanski, L., Bednarz, T., Gureyev, T., Murray, L., Huang, E., Taylor, J.: Applications of heterogeneous computing in computational and simulation science. In: 2011 Fourth IEEE International Conference on Utility and Cloud Computing (UCC), pp. 382–389 (December 2011)
26. White, B.S., McKee, S.A., de Supinski, B.R., Miller, B., Quinlan, D., Schulz, M.: Improving the computational intensity of unstructured mesh applications. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, pp. 341–350. ACM, New York (2005)
27. Keutzer, K., Massingill, B.L., Mattson, T.G., Sanders, B.A.: A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects (2010)
28. Batagelj, V., Mrvar, A.: Pajek-program for large network analysis. Connections 21, 47–57 (1998)
29. Klamt, S., Stelling, J.: Combinatorial complexity of pathway analysis in metabolic networks. Molecular Biology Reports 29(1), 233–236 (2002)