Adaptive Libraries for Multicore Architectures with Explicitly-Managed Memory Hierarchies

Konstantin Nedovodeev
Institute for High-Performance Computer and Network Technologies,
State University of Aerospace Instrumentation
Saint-Petersburg, Russia
parallelgeek@gmail.com

Abstract

Programming of commodity multicore processors is a challenging task and it becomes even harder when the processor has an explicitly-managed memory hierarchy (EMMA). Software libraries in the field of matrix algebra try to keep pace with this challenge by using the dataflow model of computation and constructing tiled algorithms. A new approach to high-performance software library construction is proposed, which moves scheduling decisions to compile-time and is portable between different EMMA platforms. Performance and scalability analyses both demonstrate promising results. Experiments demonstrate near linear speedup on a synthetic multicore architecture, incorporating up to 16 working computational cores. Performance of a generated code is competitive with vendor BLAS implementations for the Cell processor.

Index Terms: explicitly-managed memory hierarchy, adaptive library, BLAS.

I. Introduction

It is widely accepted that multicore programming is a hard problem. While creating a program we need to subdivide the whole task into subtasks, balance workload among cores, manage synchronization and manage complexity. There is a specific kind of architectures, which are even more complex to program than commodity chip-level multiprocessors (CMPs) – explicitly-managed memory architectures (EMMA) [1]. EMMA-architectures possess specific kind of problems, namely: each core has small-sized scratchpad local memories, transfers between memories have to be managed explicitly (no transparent caches), no widely accepted programming model for all representatives exist.

There are some packages exist for software libraries construction, such as Plasma [2], Cilk [3], SMPSs [4]. Some of them have been ported to the Cell processor [5]. However, the unified approach to software library construction for EMMA-architectures is still lacking.

The packages considered are best suited for those programs which represent tiled algorithms [6]. Each tile is a continuous chunk of data representing the part of blocked matrix. Such algorithms are used in linear algebra software packages, such as LAPACK [7]. It should be mentioned that the core functionality of LAPACK and other linear algebra packages is incorporated into a BLAS library [8], which is a key to achieving high performance in these packages.

With the financial support of the Ministry of Education and Science of the Russian Federation

The rest of this paper is organized as follows: in section A we concentrate our attention on EMMA architectures and their distinguishing features. Section B is dedicated to the description of an approach presented in this paper. In section C synthetic processor model construction is described. Section D contains library subprogram workflow description. Analysis of modeling results is presented in section E.

II. Main part

A. EMMA architectures

EMMA architectures incorporate heterogeneous cores: one control core, several (up to 8) computational accelerators and one or several transfer engines (TE) [5, 9, 10]. Each computational core has its private local store (LS) which is directly accessible via an instruction set. The local store is a small-sized scratchpad memory (not cache), that is why processing large data is a complex task, demanding for TEs involvement. Each TE should be programmed either by using specific instructions (e.g., for the Cell processor) or by using specific data structures, called "tasks", residing in memory, describing transfers. Main bottleneck is slow channels between common store and local store memories.

In architectures, which have more than one computational core there is a possibility of transferring data directly between local store memories of different computational cores. The aforementioned fact leads to the problem of distributing data transfers among channels so that the total time to solution be the smallest possible. TEs could work asynchronously with other components. Therefore, the maximum possible transfer hiding is of high demand.

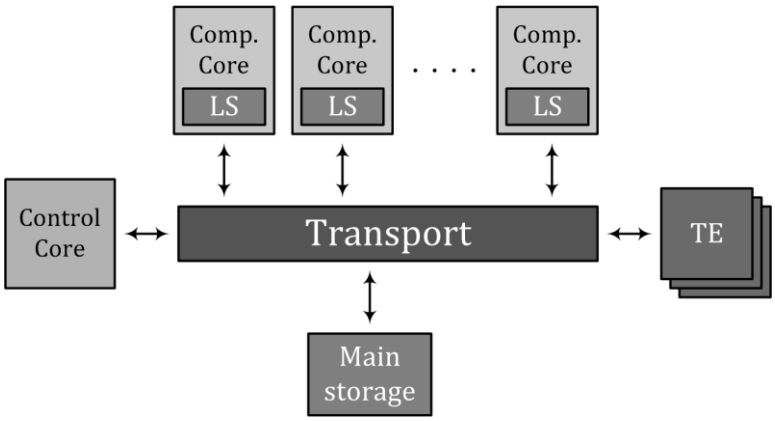

Fig. 1. The EMMA architecture flowchart

Many EMMA architectures exist these days, namely: IBM Cell processor [5], TI OMAP [9], Atmel Dyopsys [10]. This paper describes a unified approach to solve the aforementioned problems automatically for tiled algorithms; examples of algorithms will be presented later.

B. The approach description

Existing software packages such as Cilk [3] or StarS [11] are ported to the Cell procesor and consider a processor as a symmetric multiprocessor with a specific management of inter-module transfers, thanks to the instruction set support and to the fact that each computational core has a dedicated TE. Such an approach lacks unification

for those architectures, which do not have dedicated TE for each computational core. Furthermore, dynamic nature of the resource allocation [3, 4] makes it less profitable for embedded systems, which demand high performance on data sets of smaller size. Main workflow of such packages is presented in Fig. 2.

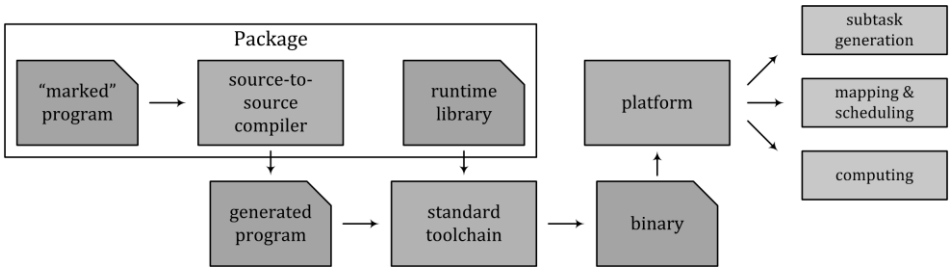

Fig. 2. Existing approaches' workflow

User has to write the program in terms of subtasks and point out which data chunks are input, output or changing in place [3]. After that a special compiler transforms this program into a new one, containing specific runtime library calls for managing pool of tasks, mapping tasks to computational cores, scheduling computations and managing synchronization. This approach leads to a significant runtime penalty for data sets below 2K elements [12].

A new approach was proposed by the author [13]. This approach extracts mapping and scheduling tasks from the runtime and performs those steps at compile-time (see Fig. 3).

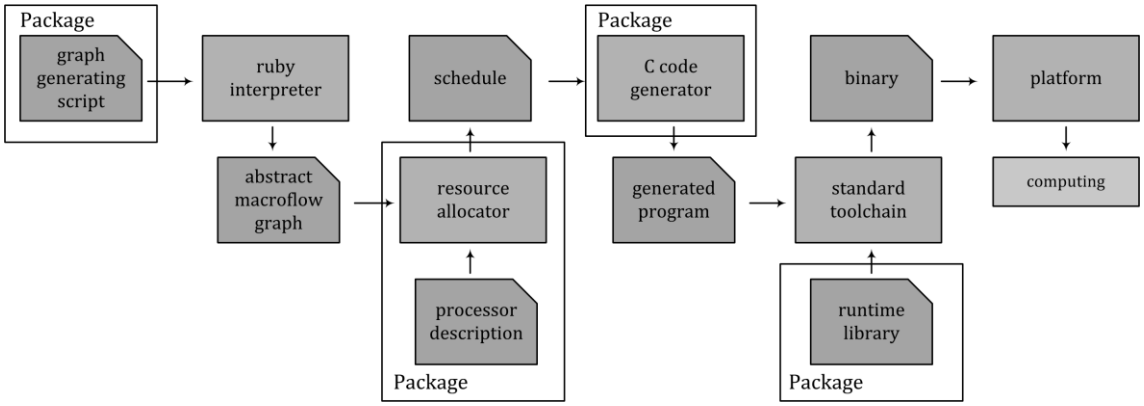

Fig. 3. Proposed approach workflow

The approach is based on a static macro-flow graph generation and resource allocation, using a processor description. At first, the programmer writes a Ruby-script, which incorporates information about how to construct an abstract macro-flow graph. Each script is specific for the type of generated programs. After that, the programmer has to write computational microkernels (not shown in fig. 2, 3). In case the runtime library is compatible with the processor model, on which the program is to run, and there is a processor description in question, the toolchain is ready to be used on this platform.

An example of a macro-flow graph is presented in figure 4. Nodes of a macro-flow graph are called "actors". All nodes have simple "firing rules" according to [14]. The bigger nodes correspond to computational microkernel calls, while the smaller ones

represent data transfers. While other researchers use a widespread representation with only “computational” nodes, the use of specific nodes for representing data transfers allows us to distribute data transfers in a more flexible fashion and estimate the memory footprint size for allocating TE tasks.

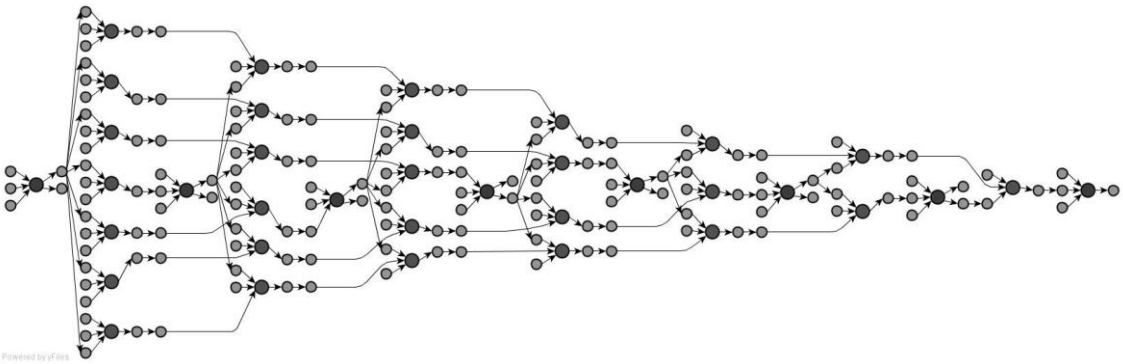

Fig. 4. An example of a macro-flow graph

The scheduling algorithm is composed of three stages: computation mapping and scheduling, transfer mapping and scheduling, memory allocation. Computation mapping and scheduling is itself a list scheduling algorithm, which uses a “greedy” ALAP [15] heuristic for mapping microkernel calls to computational cores. The list of corresponding actors is sorted using a criterion, which tries to move closer those actors, which “make heavy transfers”, and preserve the dependencies. Transfers are mapped to TEs according to their workload and the algorithm tries to “hide” transfer time “behind” computations. Transfer channels between local stores of different cores are used as much as possible. In case resulting data could be transferred through the local store, the corresponding actors are “marked” and TE tasks are not created. After computational actors have been mapped to computational cores the macro-flow graph is restructured. For each direct transfer there is a single actor and one TE task. For the transfer through the local store there is a single “marked” actor and no TE task. For each transfer through the common storage there are two actors and two corresponding TE tasks. The latter happens when the local store could not hold the result without overflowing memory.

Explicit memory allocation stage prevents data corruption and local store overflow. Using a complete macro-flow graph, we could compute maximum storage requirements, allocate buffers and spread data chunks. A multi-buffering scheme is used for interleaving data transfers and data processing inside the local store. While single buffer is used by the microkernel, others are used for exchanging data with other computational cores and the common storage. Temporary buffer is allocated inside the common storage in order to remove additional WaR and WaW dependencies and make scheduling decisions more flexible.

C. Synthetic processor models

The aforementioned approach has been implemented in a SAMPL [13] software package. By now, this package provides an ability to generate subprograms for a domestic “Multicore” family of processors, namely for the MC24, MC0226. All needed information about these processors is incorporated into the runtime library, the code generator (see Fig. 3) and the microkernels. Maximum number of computational cores

for the processors of this family is four. Porting the SAMPL package to the Cell processor, which has 8 computational cores, leads to substantial work. In order to perform a scalability analysis two synthetic processor models has been constructed.

For timing characteristics of microkernels to be the same, each processor model incorporates the same computational cores, as MC0226, having VLIW architecture and working with the same core frequency of 80 MHz. Main bottleneck of a considered architecture is a common store – local store transport channel. Each platform, incorporating a processor, may be characterized by the following ratio:

$$k = \frac{\beta}{p_{peak}},$$

where β is a throughput of the slowest channel between the common store and any of the local stores and p_{peak} is an aggregate peak performance of all computational cores. In case the Cell processor works at 3.2 GHz, the k_{cell} is the smallest one among considered architectures and $k_{cell} \approx 0.1342$. Peak performance of a computational core in a "Multicore" processor is 160 MFlop/s (SIMD mode not considered). So, to synthesize a processor model having 8 and more computational cores we need to multiply the k_{cell} by an aggregate peak performance of all computational cores. In case of taking the smallest k we make a conservative decision. The throughput of local store to local store channels in each synthetic processor model stays the same as in the MC0226 processor, namely 199.3 MB/s. Each computational core has Harward memory architecture, the data memory is split onto two memory modules. Each data memory capacity is 64KB. The program memory has 16KB. The local store size imposes an upper bound on the tile size. By now only one TE is allowed, communication parallelism is not considered yet.

D. A program workflow

Each generated program works stage-by-stage, like in a BSP model [16]. Each stage has finishing barrier synchronization. During each stage there could be some computations and communication. Each computational core could perform only one microkernel call per stage. Program preemption during any of the stages is forbidden. Computations and communication may interleave because TEs work asynchronously and local memory is multi-buffered. Code loads to the local stores are performed during program execution. All the parameters needed for performing each microkernel call are packed into consecutive data chunks and transferred to the local stores right before each call. There are no computations during the first and the last stage. The first stage incorporates only code, parameter and data loads; the last stage only data is transferred back to the common store.

Fig. 3 is simplified, because each stage incorporates TE tasks creation. Each TE has corresponding double-buffered area, one buffer is filled with new tasks, while the other one is used for fetching and processing previously created ones. TE task creation is a lightweight operation, because all the information about the transfer is computed at compile-time. So we only need to format this information so that any TE could interpret it.

E. Simulation results

In order to perform a scalability analysis two synthetic processor models have been used. They've been called MC0826 and MC1626; the first two digits denote the number

of computational cores. The rules used for construction of these models were presented in section C.

Two BLAS subprograms were taken in a consideration: sgemm and strsm [8]. The first subprogram is a good candidate for parallelization; because the structure of data dependencies between subtasks is relatively simple (each tile of a resulting matrix is a result of a simple chain of linearly connected invocations of the same operation). The strsm subprogram has a more interesting structure of dependencies. The macro-flow graph for computing single tiled column of a resulting matrix is presented in fig. 4. In general case there are several tiled columns and thus – several weakly-connected components (WCC)[17] in a macro-flow graph.

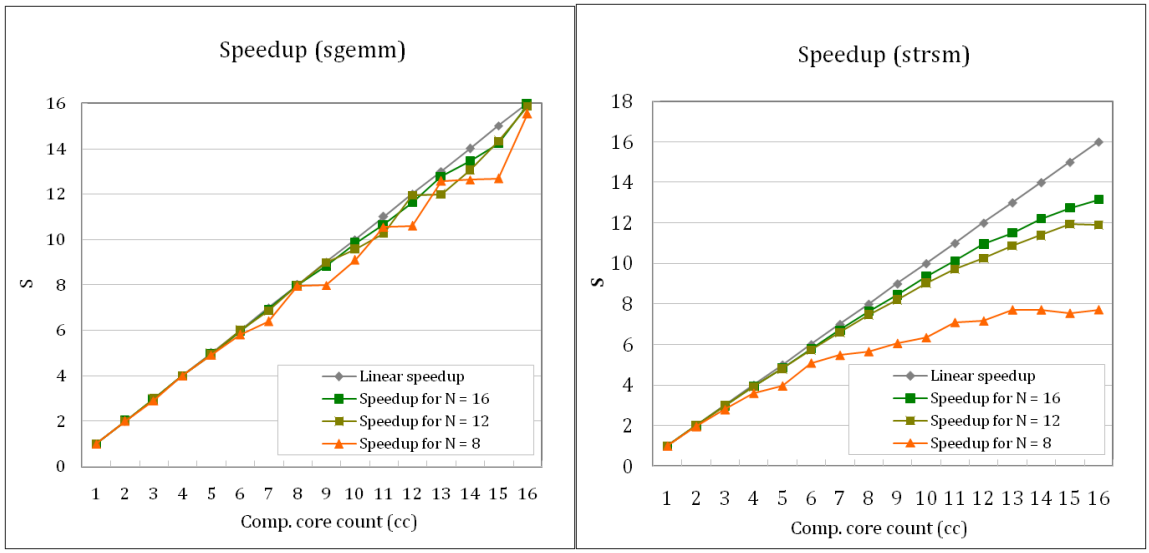

Fig. 5. Scalability of the sgemm and strsm BLAS programs

The first experiment series was performed using the MC1626 processor model. Both programs were running on data sets, having N tiles in each dimension. The number of computational cores (cc) varied from 1 to 16. Simulation results are presented in Fig. 5.

As can be seen, sgemm scales well, but its speedup oscillates. Oscillation reduces when N value increases. The oscillation takes place because of the computational workload imbalance. When N increases, the number of WCC increases also and the total makespan becomes bigger. That's why the influence of an aforementioned factor reduces with an increase of a data set size. The strsm program behaves differently. When the data set becomes bigger, the speedup increases and the curve moves closer to a linear speedup. Such a behavior stems from the following consideration: when the data set size increases, the WCC-related computational parallelism level increases also (see Fig. 4), so it's more simple for the scheduling algorithm to load the computational cores.

The second series of experiments was performed using strsm program only on the MC0826. The choice of a processor model having less number of cores made simulation time and results processing time much shorter. Program performance is presented in Fig. 6.

As can be seen, at the maximum number of tiles (N = 16), performance of the program approaches sgemm microkernel performance. The tile size (NB) was chosen equal to 38 elements.

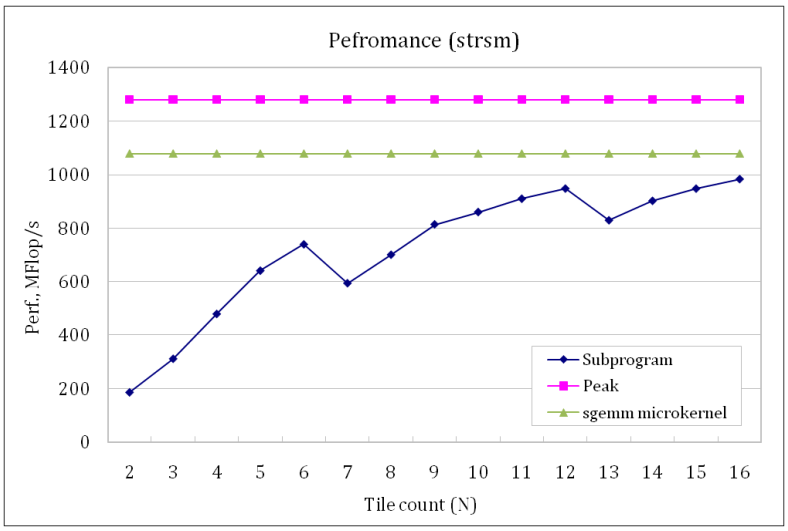

Fig. 6. Performance of the strsm BLAS program

The number of cores was 8 and stood the same in each of the experiments. Performance downfalls are caused by the fact that each scheduling step involves several weakly-connected components to be scheduled. In these experiments the number of components (wcc_s) was equal to 6 thus the points of downfalls are the points where one additional WCC is scheduled. A decrease in falling takes place when the data set size increases.

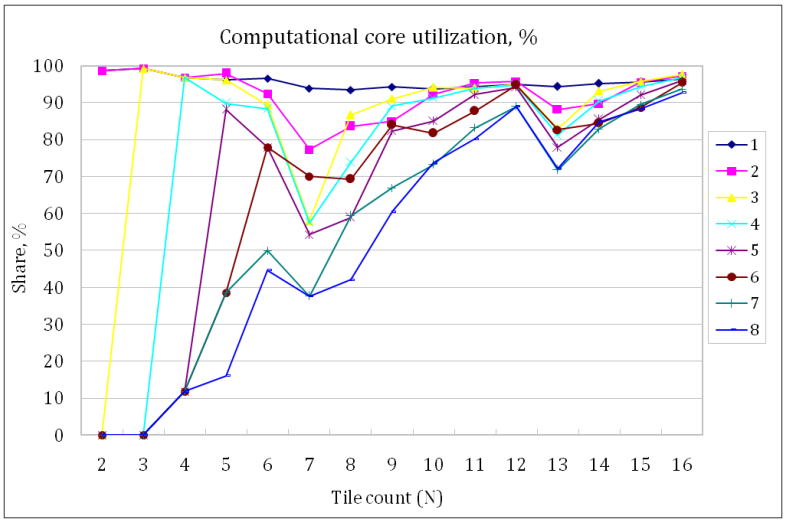

Fig. 7. Computational core utilization

Fig. 8. Heatmap of a schedule for the strsm BLAS program

Fig. 7 demonstrates an estimation of how much time do the computational cores perform the microkernel calls. It can be seen that at least one core is working almost all