On architecture for the future *petascale* computing.

Luděk KUČERA [1],
Charles University & Czech Technical University, Prague, Czech Republic

Abstract. The supercomputers listed in Top500 and running the HPCG benchmark are usually not able to use more than 1-5 % of their peak computing power, and no system is able to achieve 1 PFlop/s. The paper shows that the Conjugate Gradient algorithm without preconditioning requires only weak bandwidth of interconnect links, but, being a problem of low arithmetic intensity (flop/byte ratio), it would need the memory bandwidth about 20 times greater than available.

Based on this, the paper investigates architecture features of a prospective cost effective processor with the memory bandwidth matching the computing power, when solving problems of low arithmetic intensity.

Keywords. processor, memory bandwidth, HPCG benchmark, petaflops, active memory subsystem

1. Introduction

The exascale (10^{18} Flop/s) computing is the leading dream and goal in HPC. Expected originally for 2016 ([2]), it is still before us, the largest system is Sunway TaihuLight (0.125 ExaFlop/s peak), U.S. Sierra and Summit are expected to come with 0.15-0.3 ExaFlop/s.

The most frequently used benchmark used to rank supercomputers is LINPACK. This benchmark is essentially the Gaussian elimination used to solve a system of linear equations and it was selected because many problems solved on supercomputers come from Numerical Linear Algebra. LINPACK results correlate well with the peak computing power (the LINPACK performance is usually 65-75 % of the peak), which makes LINPACK very popular.

However, LINPACK is a direct solver applied to dense matrices, while Linear Algebra problems solved in practice are usually sparse and very large (which prevents using direct methods). This is why a new benchmark, HPCG, was introduced in 2014. HPCG is an implementation of Conjugate Gradient Method, one of the principal iterative method of Numerical Linear Algebra, and hence it is "a method from practice".

Surprisingly, the HPCG efficiency is extremely low - only 5.3 % for the best computer in the HPCG list that also appear in Top500, below 2.5 % for the remaining systems in HPCG Top10, and only 0.4 % for Sunway TaihuLight, see [3] and [6].

[1]Corresponding Author: Luděk Kučera, Charles University, Prague, Czech Republic; E-mail: ludek@kam.mff.cuni.cz

It turns out that no existing computer is able to run the HPCG benchmark faster than 1 PFlop/s, the best being the Japanese K-computer with about 0.6 PFlop/s. From this point of view, we are still waiting a supercomputer that will open the *petascale* era.

It is therefore important to identify reasons for this poor behavior and to suggest ways of improving the sparse matrix efficiency of the recent supercomputers.

In Section 2 we will analyze communication requirements of the CG algorithm to find that very low bandwidth of interconnect links is sufficient, but the principal reason of poor efficiency mentioned above is that the memory-processor interface bandwidth is much lower than needed, when the top recent processors are used. Due to space limitation, we consider Conjugate Gradient without preconditioning.

In Section 3 the results of Section 2 are generalized to low arithmetic intensity problems, and some other memory-related problems of the recent processors are discussed.

Let us recall that *arithmetic intensity* is the average number of arithmetic operations performed per one byte of data loaded from memory into processors (also called flop/byte ratio). "Low arithmetic intensity" will be often shortened as "lowain".

In a narrow sense "low" arithmetic intensity means for us about 0.25 - the arithmetic intensity of a sequence of FMA (fused multiply add) operations of the form $R \leftarrow R + XY$, where R is a register, X is a value in a register or the cache, and Y represents a stream of unrepeated double precision floating point numbers coming from the external memory. Since a FMA instruction is usually counted as two arithmetic operations, we have 2 operations per each 8 bytes (one DP number) loaded into the processor.

In a wider sense "low" means any small constant (up to, e.g., 1-2). In particular examples, we will always consider the arithmetic intensity 0.25, but it will be obvious, how to modify the results to another (higher or lower) value.

Finally, in Section 4, the observations in Section 3 will be used to suggest an architecture of a *lowain* processor - a processor adapted to execute efficiently codes with low arithmetic intensity.

An important remark:
Our study essentially demonstrates that inadequate memory bandwidth (but not interconnect bandwidth) is the principal reason of poor efficiency of HPCG benchmark without preconditioning. Listeners or readers often conclude that this problem is an oversimplification of the real situaton that invalidates our conclusions about general lowain problems, because in practice the situation is much more complicated (the nonzero structure of a matrix is much less regular and predictable, practical problems are much more complex that SpMV multiplication or unpreconditioned CG, hierarchical structure of the computer memory is not taken into account, heat dissipation is not considered, etc, etc.).

It is true that practical lowain problems are much more complex and difficult, but this simply means that the supercomputer inefficiency when solving such more complex problems will be even worse. In other words, architecture changes we suggest to improve the ratio of the memory bandwidth and the computing power of processors (or some equivalent measures) are *necessary* not only for efficient processing of our particular problem, but for *any* lowain problem. It just might happen that the changes are not *sufficient* to solve some complex lowain problems efficiently, while nothing more is needed for the CG without preconditioning with a special matrix that is studied in this paper.

2. Conjugate Gradient Method without Preconditioning

The HPCG benchmark is an implementation of Conjugate Gradient Method (CGM) with symmetric Gauss-Seidel preconditioning. Due to space limitation, our goal to find reasons of poor HPCG efficiency is simplified by analyzing the CGM without preconditioning, but keeping the special form of the HPCG matrix. As already mentioned, we will show that the interconnect bandwidth requirements of the problem are moderate, but the memory bandwidth of the present top processors does not match their computing power, which limits the efficiency of CGM without preconditioning to values below 5 %.

The Conjugate Gradient algorithm without preconditioning, see e.g. [7,9], uses a combination of three matrix/vector operations: a scalar (dot) product of two vectors, computing the vector $\mathbf{u}_1 + \alpha\mathbf{u}_2$, and a product $\mathbf{Mv}$ of a symmetric positive definite square matrix $\mathbf{M}$, usually very sparse, with a vector $\mathbf{v}$. The first two operations being computationally very simple, we restrict our attention to the product $\mathbf{Mv}$. We will drop the requirement of symmetry and positive definiteness of the matrix $\mathbf{M}$, which is needed for convergence of the CG method, but does not make computing product $\mathbf{Mv}$ any simpler.

For simplicity and because of the space limitation, we restrict ourselves to the particular type of matrices, used in the HPCG benchmark and defined in the next subsection.

2.1. HPCG Matrix

The matrix analyzed in the present paper is that of the HPCG benchmark. Consider a cubic 3D grid $\mathbf{G}$ with $N = n^3$ nodes. *Nodes* of the grid have form $u_{i,j,k}$, where $0 \le i, j, k < n$ are integers. Nodes can be linearly ordered so that, e.g., the index of $u_{i,j,k}$ is $n^2 i + nj + k$.

Two nodes u_{i_ℓ,j_ℓ,k_ℓ}, $\ell = 1,2$, are adjacent if $|i_1 - i_2| \le 1$, $|j_1 - j_2| \le 1$, $|k_1 - k_2| \le 1$.

An oriented pair of adjacent nodes is an *edge*. Edges are labeled by real numbers (i.e., double precision floating point numbers).

Such a grid defines an $N \times N$ sparse matrix $\mathbf{M}$ as follows: if two different nodes U_1 and U_2 of the grid with indices I and J are adjacent, then $\mathbf{M}_{I,J}$ is equal to the label of the edge (U_1, U_2), otherwise $\mathbf{M}_{I,J}$ is zero. The diagonal of $\mathbf{M}$ is arbitrary.

When multiplying a matrix $\mathbf{M}$ and a vector $\mathbf{v}$ to get the product $\mathbf{w} = \mathbf{Mv}$, we will store the I-th elements of both $\mathbf{v}$ and $\mathbf{w}$ and the non-zeroes of the I-th row of $\mathbf{M}$ in the node of the grid $\mathbf{G}$ of the index I.

Note that $\mathbf{M}$ obtained in this way is very sparse - each row contains at most 27 nonzeroes (including the diagonal element). Obvious modification of the definition of adjacency of grid nodes can give rise to, e.g., matrices with 19 or 7 nonzeroes per row.

2.2. Assignment of matrix/vector elements

Our first goal is to show that interconnect is not the reason of poor HPCG efficiency of recent computer, low bandwidth links are sufficient to implement interconnect.

The topology used for our interconnect analysis is a 3D cubic mesh. When solving our problem on a parallel computer with different topologies of the processor and core interconnects, it is possible to embed the 3D cubic mesh into the particular interconnect topology using standard and well known methods.

Let s and t be integers, $r = st$. Consider a 3D mesh with r^3 nodes of the form $c_{i,j,k}$, where $0 \le i, j, k < r$. Two mesh nodes c_{i_1,j_1,k_1} and c_{i_2,j_2,k_2} are adjacent if one of the

numbers $|i_1 - i_2|$, $|j_1 - j_2|$, $|k_1 - k_2|$ are 1 and the remaining two are 0. (We reserve the term "grid" for the 27 point stencil 3D cubic grid defining the HPCG matrix, and "mesh" for a 7 point stencil 3D cubic grid defined in this paragraph).

We will call elements of the mesh "cores". The mesh is subdivided into t^3 cubic submeshes, each containing s^3 cores. The submeshes will be called "processors". The mesh and its subdivision describe in a natural way interconnect topologies (at on-chip and system levels) of a parallel computer with many-core processors.

Now, let N, n, $\mathbf{G}$, $\mathbf{M}$, and $\mathbf{v}$ be as in the preceding subsection, and suppose for simplicity that r, the dimension of the core mesh, divides n, the dimension of the matrix defining grid; denote n/r by m.

The nodes of the grid map in a natural way to the cores of the mesh: a node $u_{i,j,k}$ is mapped to $c_{i/m,j/m,k/m}$. In other words, since r divides n, the matrix defining grid can be subdivided into r^3 subgrids of m nodes, and each subgrid is mapped into the corresponding core. Recall that elements of the matrix and the vectors taking part in the computation of the product $\mathbf{w} = \mathbf{M}\mathbf{v}$ are distributed to nodes of the grid, and we will suppose that an element that is associated with a grid node U is stored in the local memory of the core to which the node U is mapped.

2.3. Interconnect

When computing the matrix product $\mathbf{w} = \mathbf{M}\mathbf{v}$, each core c has to consider all grid nodes u that are mapped into it, and to compute the sums $\sum_q \ell_q v_q$, where the sum is over all grid edges q starting in u, ℓ_q is the label of the edge q, and v_q is the value of the element of the vector $\mathbf{v}$ associated with the grid node that is terminal node of the edge q.

If a grid node u is an internal node of the subgrid of the core c, i.e., each edge starting in u terminates in a node that is mapped to the same core, then all information necessary for the computation in the previous paragraph is stored in the local memory of c.

On the other hand, if u is on the boundary of the subgrid (i.e., not in its interior), it would need information about elements of the vector $\mathbf{v}$ that correspond to nodes that are mapped to different cores and, conversely, the information about the element of the vector $\mathbf{v}$, associated with u will be needed by another core, to which u is not mapped.

In order to hide the time necessary to exchange information about elements of $\mathbf{v}$ among cores, it is convenient to proceed as follows:
Each core computes first the elements of the product vector $\mathbf{w}$ for interior nodes of its subgrid (no data from other cores is needed) and, *simultaneously*, it sends the vector $\mathbf{v}$ elements, corresponding to boundary nodes, to cores that need them.

Computing one element of $\mathbf{w}$ requires 27 FMA operations (fused multiply add). Thus, $27(s-2)^3$ arithmetic operations are needed to process interior nodes of subgrids.

In the same time, a core needs to exchange s^2 vector $\mathbf{v}$ elements with each of its neighbors in the interconnect mesh, advantageously using one long message.

Thus, if $\ell + s^2\rho \leq 27(s-2)^3\alpha$, where ℓ is the latency of inter-core message, ρ is is time to send one number over the interconnect link, and α is time to perform one FMA operation, then all data needed to process boundary nodes of the subgrids are available before this computation starts.

In fact, we have to be more careful: nodes that are on the perimeter of the faces of subgrids of particular nodes need information about vector $\mathbf{v}$ elements that have to travel 2 hops in the mesh of cores, and 8 vertices of each subgrid also needs one vector $\mathbf{v}$

element that is 3 hops from its destination in the core mesh. However, if we proceed the subgrid boundary nodes so that interior points of faces of the subgrid (18 neighbors in the subgrid including itself) are processed first, then the nodes with 12 neighbors in the subgrid and finally 8 vertices of the subgrid, longer traveling time of these elements can be hidden in the time for computation of the previous category of nodes.

If, in order to get a rough estimation, we ignore the latency and suppose that $2 << s$, then we get $\rho \leq 27s\alpha$. In practice, it is usually quite easy to satisfy the bound.

For illustration assume that the total memory of a system with 256 thousands of cores is 1 PB (the memory size is typical and the core number representative for supercomputers from the top ten of Top500, see [13]). One row of the matrix contains 27 non-zeroes, i.e., (assuming DP numbers) we need $(27+2)*8 = 232 < 2^8$ bytes of memory for one grid node. Since 1 PB is about 2^50 bytes, there is space for about 2^42 grid nodes, which gives $n = 2^14$. 256k cores is about 2^18cores, which gives $m = 2^6$, $s = n/m = 2^8 = 256$. This implies that the size of core subgrids will be 256^3. Assuming, e.g., the core clock frequency 2 GHz and the core vector unit performing 16 FMA operation per cycle (as it is for Intel Xeon Phi v200), i.e., one operation each 1/32 ns, our rough estimation gives $27 \cdot 256/32 = 216$ ns as the time necessary to transfer one number over an interconnect link, which (in the case of double precision numbers) corresponds to 3.375 ns per bit, which corresponds to about 0.3 Gb/s.

However, this is just the necessary link bandwidth between two cores, but we assume that a processor has s^3 cores, and we are more interested in combined bandwidth of links connecing chips. The 3D mesh of cores is divided into cubic submeshes containing s^3 cores and assign cores so that each submesh is contained in one processor chip. In this way, each chip is interconnected with 6 other chips (those adjacent to its faces) and a link to a neighbor chip consists of s^2 core-to-core links, which gives its required bandwidth.

If, in our particular example, the number of cores in one processor chip is $64 = 4^3$, then one chip-to-chip link combines $s^2 = 16$ core-level links, and hence the required bandwidth is about 4.8 Gb/s, which is well in the state of the art.

Our analysis suggests that the communication among processors is not a limiting factor for lowain grid computations. Intuitively, this is because the amount of computation is proportional to the volume of subgrids assigned to cores or processor chips, while the amount of communication among cores/chips is proportional to the surface of the subgrids, and the ratio surface/volume is decreasing when subgrids get larger.

2.4. Memory Bandwidth Limitation

Let us first estimate the memory-processor traffic during computation of the product $\mathbf{w}$ of a HPCG matrix M defined in the subsection 2.1 and a vector $\mathbf{v}$. We assume that the matrix M and the vector $\mathbf{v}$ are initially in the external DRAM memory, and the resulting vector $\mathbf{w}$ should eventually be stored there, too.

If the grid, defining the matrix $\mathbf{M}$ has $N = n^3$ nodes, then the matrix has $27N$ nonzero elements and the vectors $\mathbf{v}$ abd $\mathbf{w}$ have each N elements. This means that at least $29N$ numbers cross the boundary between the processors and the memory of the computer. This is the information lower bound that does not reflect the fact that each vector $\mathbf{v}$ element is used 27 times and if the caches of cores are not suffiently large, it might happen that at least some vector elements are loaded more than once. Neither we consider the vector $\mathbf{v}$ elements that processors exchange among themselves, because this effect

occurs only on the boundaries of the subgrids of cores and it is negligible with respect to the subgrid volumes.

If the number of processors of the system is P and the memory bandwidth of any single processor (measured as the number of transitions of double precision floating point numbers per second) is B, then we need time at least $t_m = 29N/(PB)$ to load all elements of **M** and **v** into processors, and to store elements of the vector **w** in the memory.

Computing one element of the resulting vector **w** can be implemented as a sequence of the following steps:
CLEAR a register R;
perform 27 times a FMA operation $R \leftarrow R +$ matrix element $*$ vector element,
STORE the value of R.

Since FMA instruction is considered to be two arithmetic operations (add and mpy), computing the product **Mv** means executing $54N$ arithmetic operations. If the computing power of any single processor is Q, then the time to perform so many operations would be $t_e = 54N/(PQ)$, if all processors were working at their peak computing power.

If $E = t_e/t_m = 0.93B/Q < 1$, which is the case for recent computers, then $100E$ tells us how many percent of the peak computing power is efficiently used for computation.

The computing power of the newest Intel Xeon Phi v200 "Knight Landing" is about 3 TFlop/s, Nvidia Tesla 100 is even more powerful, about 5.2 TFlop/s. On the other hand, the combined memory bandwidth of the Intel chip is about 600 TB/s, which is about 75 Gtransition/s, where "transition" means transfer of one dual precision floating point number between the memory and the processor. The memory bandwidth of the Nvidia chip is about 720 GB/s, which is about 90 GT/s. The formula of the preceding paragraph says that the Intel chip can not compute the product of a HPCG matrix and a vector better than at 5 % of the peak, and the limit for Nvidia chip is less than 4 % of the peak. The processor SW26010 of the most powerful supercomputer to date, Sunway TaihuLight, has the peak computing power comparable to the Intel KNL, but its memory bandwidth is about 5 times lower, and therefore its upper bound to lowain efficiency is below 1 %.

It follows that the memory bandwidth that doesn't match the peak computing power of a processor is the main and very serious limitation of the efficiency of recent top processors when applied to lowain problems and the main reason of poor performance in the HPCG benchmark.

3. Problems with Memory

The final aim of the paper is to suggest architecture feature a processor that is specialized to efficient execution of low arithmetic intensity algorithms. The present section lists certain problems concerning the processor - memory interface that should be avoided.

3.1. Processor - memory interface

Let us consider solving problems of the arithmetic intensity 0.25, that of the HPCG benchmark. Since, as pointed out in the previous section, the memory bandwidth prevents an efficient use of very high computing power of recent processors, there are two extremes in designing a specialized lowain processor:

- *conservative* - while keeping the processor - memory interface essentially unchanged, processors are made simpler and cheaper by reducing their computing power to the extent that can be efficiently used for lowain problems.
- *aggresive* - in order to use full (or nearly full) computing power of the recent processors efficiently for lowain problems, the processor - memory interface is substantially enhanced to guarantee sufficiently high data flow;

Under such low arithmetic intensity, an aggresive modification of the recent 3-5 TFlop/s processors would require the memory bandwidth 12-20 TByte/s (96-160 Tbit/s), which is too much, about 20 times more then the best existing chips offer. On the other hand, the computing power of a conservative lowain processor with the memory bandwidth 600 GByte/s would run at 0.15 TFlop/s, which is now a very low value.

Therefore the development of efficient lowain processors would require a radical switch of research directions - an aggresive increase of the memory bandwidth, while keeping the computing power unchanged, or reduced to make chips more cost effective.

There are two possible ways of increasing the memory bandwidth: using greater number of processor I/O data lines and/or increasing the data transmission frequency.

The standard DDR DRAM's work at frequencies of 1-3 GHz. When using the data transmission frequency 3 GHz, either 32000 or even 55000 data pins would be necessary for fully efficient lowain use of the computing power of the above processors, which is far beyond the present state of the package technology.

It is more promissing to use ultrafast serial links for processor I/O. The speed up to 32 Gb/s is already within the state of the art (e.g., some Xilinx Virtex FPGA's have up to 128 transceivers working under 32.75 Gb/s UltraScale+ protocol [16], Cavium ThunderX2 ARM processors [1] (selected as processors of Bull's Mont Blanc computer) have integrated 25 Gb/s SerDes). In this case, the above mentioned processors would need about 3000 I/O data links for memory connection, which is a large number, but it is conceivable to build such a chip in the near future.

On the other hand, there are two advanced technologies that allow increasing the number of lines connecting a processor die with external memory dies. One is represented by Intel's EMIB (Embedded Multi-die Interconnect Bridge), see [11], another one is 3D stacking using TSV's (Through Silicon Via), see e.g. [10,15].

EMIB connects two or more dies, placed on a silicon interposer, by one or more layers of parallel microstrips or striplines attached to die boundaries.

As an example, the perimeter of a 25×25 silicon die is 100 mm. Assuming the line pitch 14 μm (the value that appears in the patent application [12], Fig. 2), we would get 7000 simple or 3500 differential lines to connect the logic die with the external memory.

If it was possible to use such EMIB for 32 Gb/s SerDes ([12] mentiones 1.6 GHz only), the problem of getting out the full computing power of a multiTFlop/s chip performing lowain computing would be solved. The same goal with EMIB working at 3 Gb/s would require 5 or more layers of the bridge lines or substantially lower line pitch and it is not clear whether this can be achieved in the near future.

Getting out the full computing power using 3D stacking working with up to 3 GHz via communication frequency would require a 2D array of TSV's with the pitch less than 100 μm, which would highly disturb the chip logic. Moreover, 3D stacking over a processor die would make power dissipation more difficult.

Another possibility, using optical channels, is out the scope of the present paper.

3.2. Near and Far Memory

A closer view of the recent processors that have > 0.5 TB/s memory bandwidth shows that insufficient memory bandwidth is not the only problem of efficient lowain computation. Advanced Intel processors and Nvidia GPU's achive the top memory bandwidth for a relatively small (up to 16 GB) high bandwidth DRAM that is tightly coupled with the processor (the processor and HBW DRAM in the same package), while the other DRAM (e.g., based on DDR DRAM chips) can be much larger (hundreds of GB per processor), but offers substantially smaller bandwidth. E.g. Intel "Knights Landing" processors offer the memory bandwidth of almost 500 GB/s when working with "near" HBW RAM, while only slightly less than 120 GB/s for "far" DDR4 DRAM channels.

When solving a large lowain problem with the data occupying a large part of the DRAM memory, a "near" fast RAM does not affect the efficiency too much, as most data must be accessed using slow memory channels. The optimal lowain memory architecture would offer (approximately) the same memory bandwidth for all memory channels.

3.3. Prefetch

Another memory related problem will be illustrated as a case study of implementing our HPCG matrix - vector multiplication on Intel Xeon Phi "Knights Landing". Assume that the matrix-vector multiplication code is simplified so that it captures the computational complexity of the algorithm, but it is not required to give any result: we omit clearing registers prior to computation, and storing their values afterwords, and (a big simplification) instead of multiplying matrix elements by vector elements, we multiply them by a constant that is stored in a register Q. The code is vectorized and the loops are completely unrolled. If the matrix data are stored in an appropriate way in a one-dimensional array $\mathbf{M}$, what we get is a homogenous and long sequence of instructions VFMADD231PD(R,Q,$\mathbf{M}[\dots]$), a code of a vectorized FMA instruction $R = R + Q\mathbf{M}[\dots]$.

In general, KNL processors execute one FMA instruction per cycle, but their operands must be in registers or L1 or L2 caches, while the elements of $\mathbf{M}$ are stored in the external memory. When a FMA instruction is executed, the memory handling hardware looks for a copy of the operand in the cache - in our case *always* a cache miss, because each matrix element is used only once. If a cache miss occurs, the hardware starts a complex process of looking for the data elsewhere on the chip (always fails in our case) and, eventually, loads the value from the external memory. Thus, each FMA instruction is preceded by a long operation (8-10 cycles) to get the matrix element into the cache.

In such a case, one should *prefetch* matrix elements into the cache before they are used to prevent cache misses. An optimized Intel compiler can be used with a directive `-qopt-prefetch=1-4` (4 = the most aggresive prefetch optimalization). Another possibility, again with an optimized compiler, is to insert directives into the program, e.g., `#pragma prefetch` in C++. Different combinations of both methods were tried, but the final code (inspected using the compiler directive -S) did not involve any prefetch instruction. It should be noted that prefetch directives are just hints to the compiler, which is not bound to use them, and since any matrix element is used just once, the compiler perhaps decided that prefetching any particular item would bring a negligible effect.

Another solution of the prefetch problem, manual insertion of prefetch instructions to the compiled .s code, is not considered (programmers are discouraged from using it).

This example shows that a fixed hardwired prefetch algorithms are not a universal solution to avoid cache misses, and sometimes fail even in a very clear and straightforward case. Good memory management is especially needed for memory intensive lowain problems, and hence more tools should be given to a user, who understands the code behavior, to control the data movement.

4. Lowain Processor Architecture

The principal message of the present paper is that we need two lines of high performance processors: for high arithmetic intensity problems and for low arithmetic intensity problems. The reason is economy - it is wasting of resources buying extremely high computing power when it can not be efficiently exploited, and arithmetic units just occupy space that could be used to build more sophisticated and powerful memory interface. The first processor line is already available - Intel MIC's, Nvidia GPU's and similar devices have extreme computing power and the memory bandwidth sufficient for high arithmetic intensity, even though some memory-related problems exist, see Subsections 3.2 and 3.3.

High performance processors for low arithmetic intensity problems do not exist, as we see from the results of the HPCG benchmark. The most important question is: do we really need them? In other words, how wide is the scope of practical problems that have low arithmetic complexity; would it justify research, investment and manufacture costs related to developing such chips? Would this approach be cheaper (taking into account extra R&D costs) then an inefficient use of existing processors?

Even though the question is far from being satisfactorily answered, there are articles that suggest that many *practical* problems have indeed arithmetic complexity close to that of the problem studied here, e.g., [14]; other papers lament about little success of attempts to increase *low flop/byte ratio* of important methods of Numerical Linear Algebra, e.g., [5]. Finally, the author's intuition says that standard methods of numerical simulation of physical processes (as air, water or general fluid flows, heat transfer, mechanical deformation) most likely result in lowain algorithms, because they are based on grids of very restricted locality, which in turn restricts the number of arithmetic operations performed with any data item in any iteration of the computation.

If we eventually decide that the lowain processor concept is economically viable, what would be the main features? With respect to the present and near future state of the art, and in view of the preceding, a TFlop/s lowain processor:

- does not need advanced CMOS technologies like 10 nm or even 7 nm resolution; in fact, the recent processor memory bandwidth does not even allow efficient lowain use of the existing FinFET technologies 14 nm and 22 nm;
- has extremely large memory bandwidth of 1000's of Gbyte/s due to
  - (speed) the use of serial transceivers with bandwidth of 10's Gbit/s to connect to the external DRAM memory, and/or
  - (width) the use 1000's I/O data lines implemented using advanced 2D, 2.5D or 3D memory-processor die aggregation based on technologies like EMIB (Embedded Multi-die Interconnect Bridge) or TSV (Through Silicon Vias)
- has a uniform bandwidth memory architecture, i.e., the memory bandwidth is about the same for all external DRAM memory attached to a processor.

Building specialized lowain processors using simpler planar CMOS technology to create only as much computing power as can be supported by the state-of-the-art processor-memory interface could be a cheap alternative for lowain problems, and the list of possible manufacturers would also be wider than the club of 3-4 companies that mastered 14/16 nm FinFET process. This is not a thrilling way to multi PFlop/s systems (with no efficient lowain use), but a solid and cheap path to go beyond 1 PFlop/s barrier when solving low arithmetic intensity problems like the HPCG benchmark.

Very large number of very fast memory channels with uniform bandwidth would almost surely lead to very different channels latencies and potentiate problems with automatic prefetch management. It is suggested that lowain processors use a memory controller that is user programmable, works in cooperation with the data processing algoritm, and delivers required data into its on-die buffer/global cache in "just-in-time" manner. Due to strict space limitation, the details and examples of such controllers are not given in the present paper.

References

[1] Cavium Introduces ThunderX2, http://cavium.com/ThunderX2_ARM_Processors.html

[2] DARPA Exascale Computing Study (TR-2008-13), http://cse.nd.edu/Reports/2008/TR-2008-13.pdf

[3] Dongarra, J., Heroux, M., Luszczek, P., The November 2016 HPCG benchmark list, http://www.hpcg-benchmark.org/custom/index.html?lid=155&slid=289

[4] Dongarra, J., Petitet, A., Whaley, R.C., Cleary, A., HPL, A Portable Implementation of the High-Performance LINPACK Benchmark for Distributed-Memory Computers, Univ. Tennessee, Feb. 24, 2016, www.hetlib.org/benchmark/hpl

[5] Ghysels, P., Klosiewicz, P., Vanroose, W., Improving arithmetic intensity of multigrid with the help of polynomial smoothers, Numerical Linear Algebra with Applications, vol. 19, no. 2 (2012), 253-267

[6] Heroux, M., Dongarra, J., Luszczek, P., HPCG Technical Specification, Sandia Report SAND2013-8752, Sandia National Laboratories, Albuquerque, NM

[7] Hestens, M.R., Stiefel, E., Methods of Conjugate Gradients for Solving Linear Systems, J. of Research of the National Bureau of Standards, 49 (1952), 409-436

[8] Intel® Xeon Phi™ v200 Product Family, http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[9] Kelley, C.T., Iterative Methods for Linear and Nonlinear Equations, SIAM Series Frontiers in Applied Mathematics, 1995

[10] Lau, J.H., Through-Silicon Vias for 3D Integration, McGraw-Hill, 2012

[11] Mahajan, R., et al., Embedded Multi-die Interconnect Bridge (EMIB) - A High Density, High Bandwidth Packing Interconnect, IEEE 66th Electronic Components and Technology Conf., 2016, pp. 557-565

[12] Qian, Z., Aygun, Z., X-line Routing for Dense Multi-Chip-Package Interconnects, U.S. Patent no. 8,946,900 B2, Feb. 3, 2015

[13] Top500 List, June 2017, http://www.top500.org

[14] Williams, S., Waterman, A., Patterson, D., Roofline: An Insightful Visual Performance Model for Multicore Architectures, Commun. ACM, 52, 4 (2009), 65-76.

[15] Wu, B., Kumar, A., Ramaswami, S., 3D IC Stacking Technology, McGraw-Hill, 2011

[16] Xilinx High Speed Serial, https://www.xilinx.com/products/technology/high-speed-serial.html