DATAFLOW SUPERCOMPUTING: CONCEPTS OF IMPORTANCE

Veljko Milutinović[*]

Keywords: *DataFlow, Maxeler, OpenSPL, SuperComputing*

Abstract: This paper focuses on designing and programming a DataFlow computer by modifying existing algorithms for better hardware utilization. Maxeler Technologies provides libraries for transforming Java-like time-critical code into the FPGA configuration files in compile-time, while C, C++, or FORTRAN code executes on the CPU and makes function-calls to start the calculation on the hardware. Using the tools from the manufacturer of the underlying FPGA technology, a .max file is formed and linked with the compiled .c code and the appropriate runtime routines, forming the executable for the control flow host. Since compilation is performed down to levels much below the machine code level (to register transfer level, gate transfer level, and transistor transfer level), the DataFlow paradigm offers better utilization of the chip for the program it is built for, making execution faster and power dissipation lower. The proposed technology is best used if the BigData can be partitioned into chunks of appropriate sizes. The time of DataFlow architecture is yet to come, mostly due to the facts that the FPGA technology and the compiler technology continue to improve; therefore, the amount of applications of DataFlow technology will increase drastically.

1. INTRODUCTION

Until a few years ago, one could say that all computers around belong to the category of control flow computers. In the case of control flow computers, one writes a program to control the flow of data through the hardware. No matter how fast the today's control flow computers are, and no matter how parallel they can be, the execution process is essentially slow. Each instruction has to be fetched, decoded, and executed. During the execution phase, one has to compute the addresses of data to be fetched, to fetch the data, to compute and store the result, etc… All these operations take time and the overall process can be extremely slow.

As of a few years ago, the DataFlow computers started to emerge. In the case of a DataFlow computer, one still writes a program, but not a program to control the flow of data through the hardware; instead, one writes a program that configures the hardware; so, when data come to its input ports, data just get flown through the configured hardware, and the result is generated one, two, or even three orders of magnitude faster. How much faster

[*] Veljko Milutinović is with the School of Electrical Engineering, University of Belgrade, Serbia

- that depends on the characteristics of the application. The data flow is not driven by a program, but by a voltage difference between the input and the output of the hardware; consequently, the process can be extremely fast, but also extremely power-efficient, and extremely small-size.

Many see the DataFlow approach as the most effective way to achieve the exa-scale speeds for big data applications. However, few understand that all the potentials of DataFlow can be fully achieved only if the algorithms related to exa-scale applications are properly modified. Therefore, this paper concentrates not only on how to design and program a DataFlow computer, but also on how to modify the existing algorithms for the best utilization of the DataFlow potentials.

The DataFlow approach represents a new old paradigm in computer design and programming. It is old, since the first ideas about DataFlow came all the way back in 60s and 70s by famous researchers like Jack Dennis (static DataFlow) and Arvind (dynamic DataFlow). It is new, because the enabler technology for implementation of the DataFlow hardware (FPGA) and software (OpenSPL) exists only for a relatively short time now. In other words, there was a relatively wide time gap between initial ideas and effective implementations, which is a characteristic of many important innovations.

As indicated above, the DataFlow paradigm, compared with the control flow paradigm, has three important advantages: speed, power, and size.

The speedups can go all the way up to 20, 200, or even 2000 (application dependant). The power reduction is about 20 times (clock dependant). The size reduction is also about 20 times (paradigm dependant).

On one hand, comparing different architectures makes sense only for the same set of applications and the same set of data. On the other hand, comparing different architectures makes sense only if the design complexity (of interest for theoretical studies) and/or the purchase price (of interest for commercial studies) are the same.

As far as the applications and data, the rest of this paper implies only the big data applications and big data volumes.

As far as the design complexity and purchase price, one has to keep in mind the following: (a) If the design complexity is fixed, then one obtains all three above mentioned advantages at the same time; (b) If the purchase price is fixed, one obtains the above mentioned advantages only close to all three at the same time, due to the fact that control flow computers are being produced in much larger quantities compared to DataFlow computers.

For all the above benefits to be achievable, certain conditions have to hold. Two are related to loop characteristics, two to application characteristics, and two to programmer characteristics.

Loop characteristics!

1. In essence, DataFlow technology migrates the execution of loops from software to hardware, which is an obvious method to make the loop execution faster. Ideally, the loop execution time is squeezed down to almost zero. In other words, for example, if a program takes 100 units of time to execute, and 95 units of time is spent in loops, after the program acceleration based on the DataFlow approach, the program execution time is ideally 5 time units. Consequently, only the applications

that spend minimum 95% of time in loops can obtain a speedup of about 20 times. This is in accordance with the Amdahl's law.

2. The above discussion assumes that, after migration into hardware, the loop execution time approaches zero. The major question is, how much close to zero? The answer depends on the level of data reusability inside the migrated loop. The more reusability, the better the speedup. In reality, the speedups can be expected only if the level of reusability is higher than 3.

Application characteristics!

1. The more the streaming in the application, the easier it is to overlap external communications (inherently slow) and internal processing (inherently fast).
2. Some applications do not tolerate even the smallest latency till the first partial result. Such applications may not be well suited for a DataFlow implementation.

Programmer characteristics!

1. The programmer has to be familiar with the DataFlow programming paradigm, which is not difficult to comprehend, but takes time to learn it.
2. The programmer must poses an excellent understanding of the underlying algorithm, so he knows how to modify it for the best exploitation of the DataFlow concept. This can also be achieved if a domain expert is in the team.

Due to the fact that the programming effort is somewhat higher and that the compilation time is somewhat longer (both will be elaborated later), the DataFlow technology is best used for the so called WORM (Write Once, Run Many) applications. In such applications, the cost of programming is amortized over the many runs of the compiled code, and is not an economic issue any more.

Modern supercomputers are carefully ranked using the Top 500 Supercomputer List, initiated about 20 years ago by Jack Dongarra and others [1]. Current #1 on the list is the Chinese Tianhe 2. Before that: Cray Titan, IBM Sequoya, Japanese K, etc… There is no single DataFlow machine on that list. The question is how come that was possible, if all the above mentioned DataFlow benefits really do exist?

The explanation is simple. Had the Top 500 List used a big data benchmark rather than Linpack (which is not a big data benchmark), and had the list been concerned with all three issues of importance (speed, power, and size), rather than with the speed alone, or speed and power, a data flow computer, like Maxeler, would be on the top of the list [2].

An elegant way to incorporate all three issues (speed, power, and size) into an analysis is to compare machines (for big data benchmarks) by how much speed can be obtained from a 1U box. If that is done, then a recent analysis demonstrates that Maxeler would definitely be on the top of the list [3].

Generally, the DataFlow approach can be of the course grain type, like in the early work of Dennis and Arvind [4], or of the fine grain type, like in the case of Maxeler. Looks like the fine grain approach is better suited for today's enabler technology: FPGA.

In addition to Maxeler, some other companies are trying similar approaches, but Maxeler is by far the most successful on the markets of application giants like Schlamberger,

JPMorgan, or Chicago Mercantile Exchange (CME), so the rest of the text uses examples exclusively based on the Maxeler approach [5].

In conclusion, the control flow paradigm implies that the compilation goes till the machine level code and that the execution process is performed at the machine code level. In the case of the DataFlow paradigm, the compilation goes to the levels much below the machine code, i.e. to the levels of gates and wires, so the process is executed at the GTL (gate transfer level). As indicated before, this brings benefits (speed, power, and size), but also the challenges: A different programming paradigm and orientation to WORM applications.

The WORM applications are found in sciences (papers in geo-physics report speedups of about 20 to 200), banking (where speedups can go from about 200 to about 2000), in image understanding, or in datamining from all kinds of sources, including also social and sensor networks.

2. THE CONCEPT

Essentially, DataFlow computers are accelerators. One continues to run the old program on the host machine of the control flow type. When the time comes for a time-consuming loop to be executed (the assumption is that the loop satisfies the six afore-mentioned conditions), the execution is passed to the DataFlow accelerator that is connected to the host machine via a PCI Express bus (for larger systems, the connection is best done via two busses: PCI Express and InfiniBand).

For the host and accelerator to integrate completely, one also has to download (at the host) the following additional pieces of system software: (a) The Maxeler OS, (b) The Maxeler Compiler and the related Run-Time library, and (c) The Maxeler Simulator, as indicated in Figure 1.

Figure 1 assumes that two loops have been migrated from the host to the accelerator (two kernels: 1 and 2). This does not necessarily mean that only two loops existed in the host application; this could mean that the number of loops in the host application could had been higher, but only two loops satisfied the conditions for migration into the accelerator.

For each loop, one has to write two programs: (a) Kernel and (b) Kernel Test. This means, if n loops are migrated, one has to write 2n new programs. Also, no matter how many loops are migrated, one has to write three new programs from scratch: (a) Manager, (b) Simulator Builder, and (c) Hardware Builder. Therefore, the total of 2n+3 new programs have to be written from scratch. All these programs are written in Maxeler Java, which is a superset of Java, with dozens of new functionalities (built-in classes) added on the top of the classical Java. Consequently, Maxeler Java is a new DSL (Domain Specific Language). Recently, Maxeler Java emerged into a new and a more elaborated language environment called Open SPL (Open Spatial Programming Language). Of course, a manufacturer-written "Makefile" has to be used, too.

The host can be a simple PC, or a most sophisticated supercomputer. The accelerator itself can be a PCI Express board, or a 1U box, or a workstation with several 1U boxes, or a rack with 10, 20, or 40 1U boxes, or a train of racks.

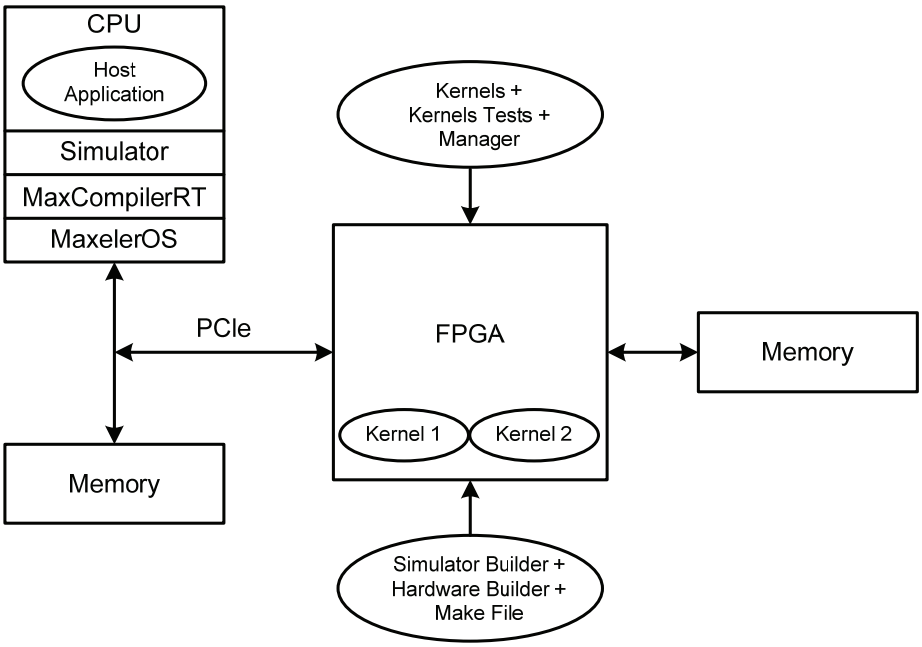

Fig. 1. Generic Acceleration Oriented Architecture.

In general, there is a 1:1 correspondence between the number of loops migrated into the accelerator and the number of kernels that one has to write from scratch. However, in reality, sometimes we have more kernels than the migrated loops, or the opposite.

We have more kernels (than the migrated loops) if one loop is described with more kernels (i.e., if the one uses the "divide and conquer" strategy). We have less kernels (than the migrated loops) if one kernel can be used for the execution of two different loops in two different parts of the host application (i.e., if one uses the "time sharing" strategy).

3. THE PROGRAMMING PARADIGM

Figure 2 represents the first introduction to the OpenSPL language Maxeler Java (MaxJ), using an example related to the computation of the moving average, which is the essential element of all algorithms based on convolution.

The MaxJ programming language includes two types of variables: (a) The standard Java variables that will be referred to here as software Java variables and (b) The DataFlow Java variables (DFVar) that will be referred to here as hardware Java variables. The software Java variables are there to instruct the compiler, so they disappear after the compilation process is completed. The hardware Java variables are there to actually flow through the hardware, so they produce the results after the DataFlow process is completed.

As indicated in Figure 2, the MaxJ programming language follows the syntax of the Java language, except for the add-ons. For example, a hardware variable is written under the quotes (e.g., "x") while it is in the environment outside the Maxeler system. Once it enters

the Maxeler system, the quotes are eliminated (e.g., x, without quotes). This is well seen in Figure 2, together with the convention used to define the appropriate hardware variables (HWVar), as well as to define the floating-point precision, exponent, and mantissa (8, 24). All other hardware variable types are defined similarly.

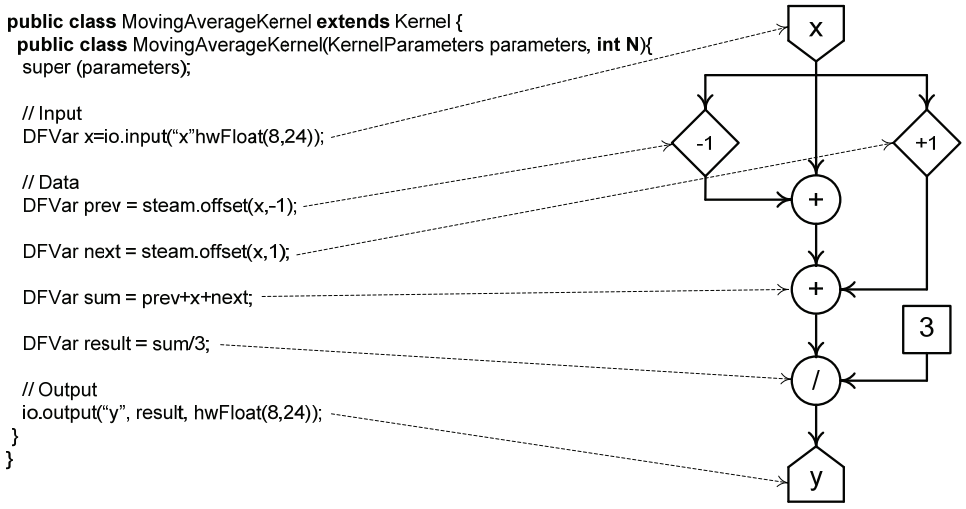

Fig. 2. Maxeler Java, the Program Code and the Related Graph.

The Maxeler compiler produces a graph, as given in the right-hand part of Figure 2. The graph can be presented in a graphical form (as indicated in Figure 2), or it can be described in VHDL (which is of importance for further processing). Further processing, from the graph level (left by the Maxeler compiler) till the binary level (needed for configuration of the FPGA circuitry), is done using the synthesis tool of the manufacturer of the FPGA circuitry used to implement the Maxeler system, which is elaborated later on.

Figure 3 describes the relationship between the hardware and software components in a Maxeler DataFlow system.

The ”for” loop from the main program in Figure 3 is assumed to be the loop to be migrated from the host to the accelerator. For that to happen, the program segment implementing the loop has to be deleted (the solid line in Figure 3), the input data for the loop have to be moved from the host memory area A to the accelerator internal memory, and after the results related to this loop are generated, they (the results) have to be moved from the accelerator internal memory to a memory area B for further processing (and eventually, at the end of the entire loop related process, back to the host). The data movement is realized by the “stream_data(device, A);” construct, which implies that the “device” has to be defined first (“device = max_open_device(maxfile, “/dev/max0”);”). As indicated by Figure 3, all these changes happen in the host code.

It is the manager code who is responsible for accepting data on the accelerator side, which (the acceptance of data) is implemented via “link(“A”, PCIE)”. It is also the manager who is responsible for moving the partial results (“link(“B”, DRAM(LINEAR)”) to the

memory area B in the accelerator (and eventually, for moving the results of the looping related process back to the host). In the cases with more kernels, the manager code is also responsible for moving of data in-between the kernels. The manager code is also given in Figure 3, as well as the kernel code, which is here repeated from Figure 2.

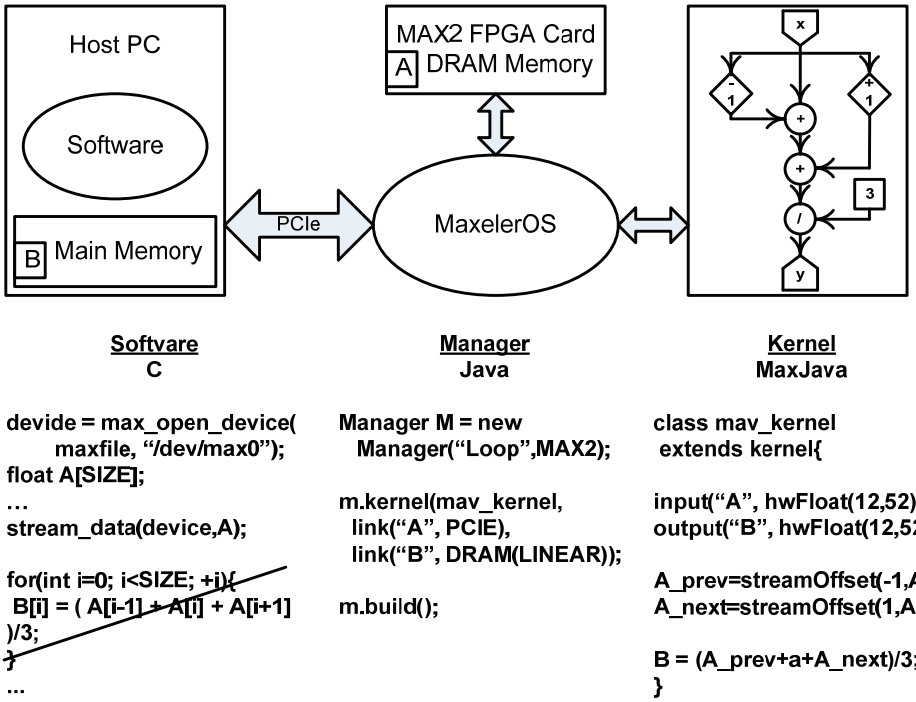

Fig. 3. Hardware/Software Relationship in a Maxeler DataFlow System.

Before the processing starts, data have to be moved, from the external memory area A, through a relatively slow PCI Express bus, which means that, in the DataFlow paradigm also, the major bottleneck is related to moving data in and out, to and from the system. Moving data to the temporary memory area B inside the accelerator system is not a problem, since it can happen at speeds that are an order of magnitude (or more) faster.

Overall, the manager code is responsible for three activities: (a) Moving data from the host to the accelerator, (b) Moving results from the accelerator to the host, and c) Moving data in-between the kernels. Consequently, compilation of the manager program could result in three different pieces of the "object" code: (a) One that is executed on the host CPU, to serve as the host-side "pillar" of the bridge connecting the host and the accelerator, (b) One that is executed on a small (for programmers invisible) CPU on the accelerator-side, and (c) One that is incorporated into the .max file that is used for configuring of the FPGA infrastructure of the accelerator.

As far as the kernels are concerned, they all are compiled into one .max file for configuring of the underlying FPGA infrastructure. The next section gives more details about the compilation process.

4. COMPILATION

Figure 4 sheds light on the compilation process and helps understand all the time constants related to the compilation process.

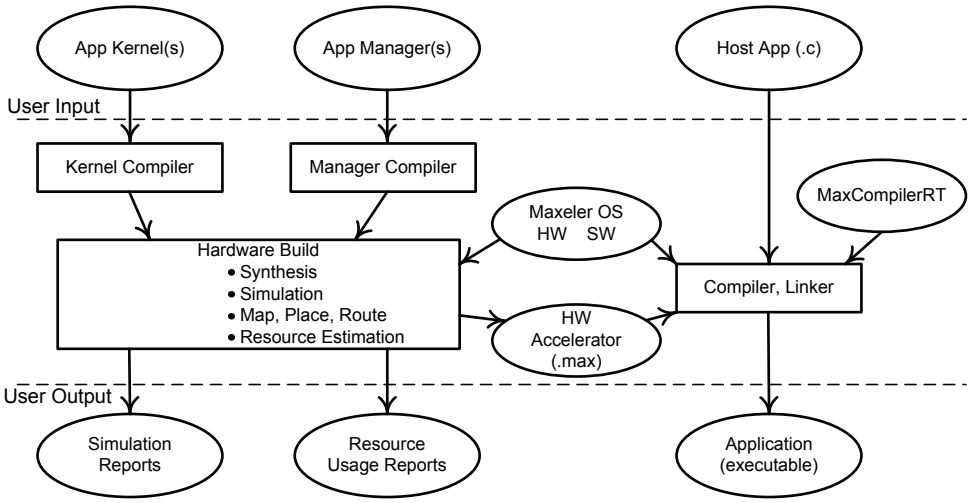

Fig. 4. Issues of Importance for DataFlow Compilation.

The compilation process is entered with: (a) One or more kernel programs (.java files), (b) One manager program (also a .java file), and (c) The host program (e.g., a .c file), as indicated in Figure 4. With the help of MaxelerOS mechanisms for building hardware, the compiled kernel and manager code are combined, and the execution graph is formed. In Figure 4, the execution graph is depicted with the horizontal line at the top of the "Hardware Build" block. The Maxeler simulator works on the execution graph level, so that level will be denoted here as the Simulator Level.

To move from the Simulator Level to the .max level, one has to use the tools from the manufacturer of the underlying FPGA technology. Once the .max file is formed, it is linked with the compiled .c code and the appropriate routines from the MaxCompilerRT (RT stands for Run Time). The final product of the compilation and linking is the executable for the control flow host.

When the execution starts on the host machine, a communication is established with the accelerator, and the accelerator is "asked" if it is configured with the related .max file. If it is not, then the accelerator is configured, by moving the .max file from the host to the accelerator. After the configuration is completed, the Big Data stream has to be activated, and the execution is to start. On the next re-run, no initial configuring is needed, unless another user was using the entire hardware in the meantime. Of course, if another user was

using just a part of the hardware in the meantime, and did not damage the configuration used by the initially mentioned program, then no reconfiguration is needed.

What are the time constants involved in the process?

These time constants are technology-dependent – the values given here are approximate and changeable. Compilation from the level of kernels to the .max level may take several hours. Compilation from the level of kernels to the graph level may take several minutes. Loading of the .max file and configuring of the FPGA hardware may take several seconds. Starting a Big Data stream may take a few milliseconds. Starting the execution of the compiled code may take several microseconds. Of course, the values of all these constants are likely to change over time, as the technology changes.

5. COMPARISONS

A crucial question is how the two concepts compare: Control flow and data flow. Both concepts subdivide into two major categories.

The control flow concept subdivides into: Multi core and many core. The DataFlow concept subdivides into: Coarse grain and fine grain.

In the case of control flow, the major issue is that the application software contains enough parallelism, so that all cores can be kept busy, and partial results appear periodically in small (multi cores) or big (many cores) chunks. In the case of DataFlow, the major issue is the delay till the first partial result on the module level (coarse grain) or the loop iteration level (fine grain).

As far as the clock speed, it is typical that the control flow clock is faster, and the data flow clock is slower, which determines the power requirements of the two approaches, as it will be discussed later.

Occasionally the researcher joke that using an N-core system is like plowing with N horses. The major question for multi-core systems (e.g., Intel) is: Which way will the horses go. For them to go into the desired direction, a plow driver is needed. We articulate our wishes in the form of a program that we load into the brain of the driver. In order to understand our wishes, the brain of the driver has to be equipped with von Neumann constructs, and in order to understand them fast, the brain of the driver has to be equipped with constructs like caches, predictors, etc…

In the same joking style, using a many core system (e.g., NVidia), is like plowing with N thousand chicken. In that case (using the CUDA programming model), more drivers are needed in the system, and each one's brain is also equipped with von Neumann constructs. In the more recent rCUDA (remote CUDA) programming model, in addition to more drivers, one dispatcher is needed, too.

The question now is what one plows with in the case that a DataFlow system is used? The answer is: With N million ants. Each ant has a back-pack for a portion of Big Data. The idea is, while traversing the plough field, that the ants generate the end result. For that to happen, one first has to configure the field (to write kernel programs) and to load the back-packs of the ants (to move data into the accelerator). What motivates the ants to move into the right direction, in conditions when no control flow program exists? The voltage difference between the input and the output of the FPGA structure moves the data!

Why is the DataFlow paradigm so fast? Because it compiles down to levels much below the machine code level; it compiles down to RTL (register transfer level), GTL (gate transfer level), and TTL (transistor transfer level), as indicated in Figure 18. Another way to look at the issue is by stating that the DataFlow paradigm uses bit serial arithmetic and internal arithmetic pipelines, which enables more computation to be done per time unit.

Why is the DataFlow paradigm so much more power efficient? Because the power dissipation is calculated using the following formula:

$$P=f(f,U^2)=kfU^2$$

where U is the voltage applied, f is the frequency applied, and k is a constant. No matter if control flow or data flow is being used, U is the same, and the constant k is the same for the same implementational technology. What differs is the frequency f. In the case of modern control flow multi core implementations, the frequencies go up to about 4GHz (e.g., Intel). In the case of modern control flow many core implementations, frequencies are lower, but still relatively high. In the case of FPGA-based DataFlow implementations, frequencies are as low as about 200MHz (e.g., Maxeler). If we divide 4GHz by 200MHz, we get 20, which is the power savings ratio most frequently mentioned in the literature.

Finally, why is the size of DataFlow implementations so much smaller? Because the control flow paradigm is based on the von Neumann architecture in which only about 5% of the chip area is dedicated to ALU area, while the DataFlow paradigm is best implemented on the top of FPGA architectures in which the more than 95% of the chip area is dedicated to ALU-type functionalities. The ratio of "more than 95%" and "about 5%" is about 20, which is the number most frequently quoted in the literature. If one likes to preserve the 20:1 ratio in size even after packaging the chips, one has to place in very small coolers and very small fans; consequently, the DataFlow implementations are noisy. The noise comes for two different reasons: (a) Smaller fans are noisier, and (b) Since coolers are smaller, the fans must rotate faster, and fast fan rotation is the second major source of fan noise.

The programming effort is not higher. The number of new programs to write is higher (2n+3), but they are straightforward to write and one does not have to worry about issues like the influence of cashes, TLBs, predictors, or memory consistency. However, it does not mean that one does not have to worry about memory hierarchy. Actually, the memory hierarchy does exist in DataFlow systems, typically two levels inside the FPGA chips, and another two levels on the accelerator board outside the FPGA chips. That memory hierarchy is to be worried about at programming time, when kernels are written.

The debugging effort is higher, since more lines of code mean more opportunity for a bug to happen. That is why a strong system development support had to be developed in packages supporting the OpenSPL concept.

Finally, the needed compilation efforts deserve special attention. As indicated before, compilation takes time and therefore applications are preferred for which one compiles once and runs the code a relatively large number of times.

6. MAXELER HARDWARE

Maxeler implementations of the DataFlow paradigm include three different options: (a) The C option, representing a full blown computer (C = computer), (b) The X option

representing an accelerator (X = accelerator), and (c) The N option for low latency LAN applications (N = Network).

The option C includes 4 DFEs (data flow elements), plus a control flow host. This option is for those who need a full blown computer, rather than just an accelerator. The reason for that may be that the user computer center does not own an InfiniBand bus, which is needed (in addition to the PCIe bus) for bigger Maxeler systems.

The option X includes no control flow host, which leaves space for 8 (not 4) DFEs – all that for the same purchase price, which makes the accelerator option much more desirable, if the user does own the InfiniBand bus.

Finally, option N minimizes the latency (which is the Achileus heal of DataFlow supercomputing) and can be used in latency intolerant applications, like trading or financial analytics, where trading data and data to be analyzed arrive via a network (e.g., Ethernet). It is obvious, in trading applications, even if one uses the same buy-sell algorithm as the competition, one has an obvious advantage due to high speeds that the data flow paradigm offers.

In the latest issue, the amount of on-board memory is 768 GB and approaching over 1 TB. This means, the technology is best used if the BigData can be partitioned into chunks of 768 GB, or 1TB in near future.

7. CONCLUSIONS

The time of DataFlow architecture is yet to come, mostly due to the facts that the FPGA technology and the compiler technology continue to improve. In addition, once the community fully realizes all the DataFlow potentials in a speed, power, and size, the amount of applications of DataFlow technology will increase drastically.

The DataFlow technology advantages are best described using the notions from Feynman [6]. Feynman claims, arithmetic and logic could be done at zero energy; communications never. The control flow paradigm follows the Von Neumann principles, which means that communications are extensive, to fetch instructions and data. The DataFlow paradigm follows the execution graph, which means that communications are minimal, just from one computational unit to the next, at negligible distances, if the compiler is smart enough.

REFERENCES

[1] Dongarra, J. J., Meuer, H. W., Strohmaier, E., "TOP500 Supercomputer Sites," The TOP500 Report, 1998.
[2] Flynn, M. J., Mencer, O., Milutinovic, V., et al. "Moving from Peta-flops to Peta-data," *Communications of the ACM*, May 2013, pp. 39-43.
[3] Dong, J., et al. "A Step Towards Energy Efficient Computing: Redesigning A Hydrodynamic Application on CPU-GPU," http://icl.cs.utk.edu/news_pub/submissions/ACMGPUHydro.pdf, June 11, 2014.
[4] Milutinovic, V., Editor, "*Advances in Computer Architecture*," North Holland, 1986.
[5] Maxeler Technologies, https://www.maxeler.com/, June 11, 2014.
[6] Feynman, R. P., "Feynman Lectures on Computation," Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.