Survey of Technologies enabling CPU Offloading presented in TDT24

Courseyear 2014, Exam 2015

Christian Chavez, chrischa@stud.ntnu.no
Student, NTNU
Trondheim, Norway

Index Terms—**GPU, GPGPU, Offloading, NTNU, TDT24, 2015, Parallel Environments and Numerical Computing.**

Abstract—**With the beginning of the end of Moore's Law [1], new challenges have been discovered in the never-ending quest to provide faster computers and applications. Manferdelli *et al.* [2] lists these new hindrances as the *power wall + instruction-level-parallelism (ILP) wall + memory wall* = The *Brick Wall*.**

Each of these walls affect parallel and High Performance Computing (HPC), and this survey paper attempts to list the most prominent technologies used in HPC, as well as some of the solutions to the *Brick Wall*, which were discussed/presented in TDT24 2014 at NTNU.

I. Introduction

WITH the introduction of Moore's Law [1], computing power increased exponentially for decades. This has been possible due to the ever-increasing amount of components (transistors) technology has been able to cram into the same processor die area.

However, this trend is coming, if it has not already, to an end, due to the physical limits of miniaturized electronics.

Hence, new challenges in continuing this exponential growth in computing power have arisen. Manferdelli *et al.* [2] introduce these new problems as three "walls", which together form the wall coined the "Brick Wall". The "Brick Wall" consists of the "Power/Heat Wall", the "ILP Wall", and the "Memory Wall". Section II describes these walls and their impacts on HPC and parallel programming in greater detail.

Section III continues the report by detailing examples on how multiple CPU cores have been utilized to bypass these walls, through utilizing multiple cores to offloading work from the main CPU/CPU-core. Byun *et al.* [3] and Newburn *et al.* [4] both offload work from the main CPU core, but they do so in two different ways.

With the introduction of GPGPU-offloading with the Intel Xeon Phi^{TM}in Section III, the report continues by details one of the most utilized GPU offloading technologies in Section IV. Section IV first introduces Nvidia's technology CUDA^{TM}, and describes how some papers have utilized it to achieve higher computing throughput of select applications.

Lastly, the report continues with introducing a second well-known heterogeneous computing language in Section V. This section introduces OpenCL^{TM}, a language/technology meant to address the ease of using multiple architectures to run the same application. Section V also introduces an optimization utilized by the LLVM compiler when writing OpenCL, relevant to multi-threaded architectures.

Finally, in Section VI we summarize the technologies (both hardware and software) used in the reports surveyed in this survey paper.

II. The Brick Wall

As mentioned in Section I, Manferdelli *et al.* [2] reports on the "Brick Wall", which they split into three "sub-walls" in their report. While these are not the only difficulties in the never-ending quest for increasing computing power, these three are the big, well-known ones that keep computing power from continuing the prediction of Moore's law.

This section lists the three and explain how they prevent the continuation of Moore's law, which predicts the exponential rise at constant cost every 18 months.

A. Power/Heat Wall

The miniaturization of components used to build CPUs has been one of the biggest driving forces, if not the biggest one, behind Moore's Law.

As miniaturization decreased the size of the components used in a CPU's die area, so did the cost per component in the CPU. As the size decreased, the number of units possible to cram into the integrated circuit (IC) increased. With the number of transistors in the CPU increased, so was the increase in clock cycles (frequency) permitted, directly affecting the number of instructions possible to execute per second. The heat generated when powering the miniaturized components, at higher frequencies, increased exponentially due to power leakage of transistors at these small sizes, and the density of components generating heat in the IC when in use.

Hence, the power demands required to keep the units cooled down to acceptable working temperatures far exceed the power demands of utilization. This gave rise to the term "Dark Silicon", coined by the strategy of turning off select subsections of the CPU's die area during utilization, so as to keep the temperature sufficiently low to function.

So far, notwithstanding major breakthroughs in processor die materials or electronics avoiding the "Heat Wall" generated by the "Power Wall", the main tactic to counter this problem has been to spread the workload across multiple cores. Often utilizing specialized heterogeneous cores such as GPUs for suited applications.

B. *Instruction-Level-Parallelism Wall*

Instruction-Level-Parallelism (ILP) is the simultaneous execution of multiple instructions in a sequential program, so long as there are no data dependencies between the instructions.

One approach to accomplish ILP has been Very Large Instruction Words (VLIW), requiring the compiler to see the opportunities for ILP in a program during the offline compilation. VLIWs reliance on the compiler to perform optimally has hindered the approach's widespread use and acceptance on HPC platform and personal use platform, which often run very diverse programs. However, on embedded platforms, typically those that only execute a few different programs, though they may need to execute them frequently and/or efficiently, has been where the VLIW shines, as there is little variety in the programs run on these devices.

While this is now changing with embedded processors in devices such as smartphones being more and more utilized for a variety of tasks, there is one other approach to ILP that has been much used on the personal and HPC platforms.

Out-of-Order execution has been a hardware optimization that permits the CPUs themselves to see and act on potential ILP in a program at run-time, namely online, as opposed to VLIW, which works offline.

O-o-O cores read their given instructions as they come in, and if they find instructions that have no dependencies among the ones loaded in cache, they use whatever free resources[1] to execute CPU instructions that the program has not yet reached.

However, while ILP works, to the degree limited by cache size and hardware, it's big limitation lies in the majority of programs being written as serial programs. Thus hindering the further speedup through utilization of O-o-O execution.

The main tactic used to counter the "ILP Wall", has been to write more parallel programs. Programs whose algorithms rely less on sequential steps, and more on discrete, independent steps, "sewn together" at critical points in the program than serial programs.

C. *Memory Wall*

Finally, the "Memory Wall" has come as a consequence of the increasing speed and throughput of the processor diverging with the accompanying required speed and throughput from memory, which has not had the same exponential growth. Wulf and McKee [5] detail the then impending "Memory Wall", and explain in their paper how, even given the most optimistic assumptions, it will come to pass with the increasing difference in throughput of execution of instructions on the CPU, and the throughput bandwidth in memory.

The memory hierarchy is an attempt to alleviate, and to a certain degree *hide* the difference in throughput bandwidth between larger, slower memory technologies (like HDDs), and faster, smaller memory technologies like caches.

[1]Such as those available when there's been a cache miss.

However, the effect of the memory hierarchy is dependent on the *correct* memory being loaded from the lower, high-capacity ends of the hierarchy to faster, low-capacity upper ones, so as to be accessible as soon as possible, if not even before execution (when the CPU needs the data located in that particular part of memory). Operating Systems (OSs) have long worked on hiding this, and in today's technology, do a good job of hiding the discrepancies from their human users on personal computers and devices.

Nevertheless, on HPC systems, which are not used for the typical and quick programs most frequently utilized on personal computing systems, the programs being run can take up to days, if not weeks or months to execute successfully (and continuously) from start to finish.

Thus, when high thermal, geological, meteorological calculations, simulations need to process amounts of data on the scale of terabytes, $8 * 10^{12}$ bits, all of that data has to pass through the fastest part of the memory hierarchy, the registers in a CPU. The registers are also the ones with the lowest memory capacity, often on the order of X registers, each with 64 bits or less, with $X < 64$ in today's most modern and common high-end CPUs. Hence, the advent of caches, and prefetching, in an attempt to have the data required in the next execution cycle of the CPU as close to the registers in the hierarchy as possible.

However, while the memory hierarchy is big and includes several very different memory technologies, the "Memory Wall" most often refers to the discrepancy between on-chip memory bandwidth, and off-chip memory bandwidth. In simpler terms, when the data is transferred between ICs.

Besides the aforementioned caches and prefetching, there are several main tactics to combat the "Memory Wall", such as the exploitation of temporal and spatial locality in caches. One other tactic is writing the programs to be more parallel, permitting the load of the memory bandwidth on the CPU to be spread across multiple CPUs, if not also CPU cores.

III. CPU Offloading

This section introduces Offloading, by first explaining the term, before explaining how parallel programs implement this, and their difficulties and limitations. After that, we summarize the paper of Byun *et al.* [3], and explain how they offload the CPU.

A. *Offloading*

Offloading is the alleviation of the workload of a program normally/otherwise run on a single agent, by having multiple sub-cores of the CPU, CPUs, or other processing units entirely execute some of the program's workload. By spreading the load, only possible with parallel programs, the total execution time of the program can be diminished, even with the bottlenecks from the "Brick Wall" affecting each CPU or CPU core (hereafter referred to as "agents", to avoid discriminating other processing units).

Rather, if we generalize a simplified optimum case, one can propose that the amount of agents P, the time spent executing the program on a single agent T_1, gives the formula for the speedup gained by executing with the load distributed across P agents to be $S_P = T_1/P$.

B. Implementing Parallelism

However, since the formula $S_P = T_1/P$ is a simplified generalization of the optimum case, it gives a wrongful representation of T_P. While a program running on P agents may approach the speedup given by T_1/P, it can never be equal to this, since then we would be running P completely independent programs. Amdahl [6] proposed that each program needs to have a sequential part, such as the summing of the local sums from each agent in a parallel programming calculating a sum to a final result.

This gave rise to *Amdahl's Law*, which states that the theoretical max speedup of a parallel program becomes the sequential part, plus the parallel part divided by a number of agents (e.g. CPUs) executing the parallel part simultaneously. Hence, Amdahl's law gives the formula $S_P \leq P/((P-1)T_S + 1)$, with P, T_1, T_P representing the same as previously given, and T_S the time it takes to execute the sequential parts of the program.

Amdahl's law gives a bleak estimate of how much parallelizing a program can help its execution time. However, this bleak estimate is with one caveat, and that is that the formula presupposes a *fixed* problem size speedup.

This did not escape the notice of Gustafson [7], who proposed a new formula for calculating the theoretical max speedup of a parallel implementation, but with *fixed* time and variable problem size speedup, as opposed to Amdahl's law. Hence, *Gustafson's Law* gives the formula: $S_P = P - T_S(P-1)$.

Thus parallelizing programs still had more benefits to give, beyond what Amdahl's law proposed, as long as they increased the problem size along with the size of P.

Sun and Ni's Law [8], [9] proposes a third way to look at speedup in parallel programs, and that is $memorybounded$ speedup.

Start by supposing that f is the portion of workload that can be parallelized, and $(1-f)$ the sequential part.

Given a memory bound function $G(P)$ representing the influence of memory change on the change in problem size, Sun and Ni's law give the following formula for calculating the speedup: $S_P = ((1-f) + f \times G(P))/((1-f) + \frac{f \times G(P)}{P})$.

If $G(P) = 1$ (our $G(P)$ being the function $\overline{g}(m)$ in Sun and Chen [9]), then the memory-bounded speedup model reduces to Amdahl's law, since the problem size is fixed, or independent of resource increase. If $G(P) = n$, then the memory-bounded speedup model reduces to Gustafson's law, which means that when memory capacity increases m times, and the workload also increases m times, all the data needed is local to every node in the system.

Thus, Sun and Ni's law directly relates to, and shows how to solve, the "Memory Wall" explained in Section II, on a parallel system with homogeneous agents.

C. Offloading through Threads

A CPU can run a program in parallel in several ways. It may have multiple cores, which each can run a process/program concurrently. Also, a CPU core may have multiple threads running concurrently, depending on the hardware available. The first report we summarize, utilizing parallel optimizations, utilizes threads and cores to enable the parallelism of their system.

Byun *et al.* [3] proposes a new parallel system: "pOSKI", which is a parallelization of the SpMV autotuning framework "OSKI". SpMV performs traditionally poorly on single-core agents, due to the irregular memory accesses and the "Memory Wall".

Parallel "OSKI" ("pOSKI"), utilizes beforehand tuning knowledge to select proper compilation parameters, loop unrolling, and variable sizes to optimize the code, though these code optimizations are also applicable on a single-core system. On the other hand, mapping blocks of a matrix to singular threads running concurrently on a CPU, using *Non-Uniform Memory Access* (NUMA) aware mapping of the matrix partitions to cores, and utilizing SIMD intrinsics to fully utilize the available ALUs to concurrently calculate more than one arithmetic operation, are parallel optimizations described in the report as utilized.

While Byun *et al.* [3] are among the more specialized papers in this survey, they do offer insights into the applicability of their parallel optimizations, noting that they do not always result in a speedup.

Their results are gathered from experiments on several different architectures and configurations, with numbers as high as 8.3x faster than the serial OSKI implementation, and as slow as 1.3x and 1.2x slower than the parallel Intel MKL and OpenMP implementations of the same SpMV on their test setups.

D. Shared Memory vs. Distributed Memory

When discussing parallel programs, it's often important to mention whether the algorithm utilized by the program is implemented with regards to shared or distributed memory.

Besides the other technologies introduced in this paper, there are two widely used technologies that we do not discuss in any other section, but that are still worth mentioning.

- Shared Memory: Shared memory is when multiple agents share the same memory during execution. Such as threads using the same process memory in a CPU core. Threads are most commonly the technology referred to when discussing shared memory, and Pthreads, OpenMP, Intel's Building Blocks$^{\text{TM}}$(TBB) are among the most used technologies to program algorithms relying on shared memory.
- Distributed Memory: Distributed memory, however, is when each agent has its own memory, inaccessible to other agents. CPUs being a good example of this model, MPI are among the most, of not the most known technology utilized when writing programs relying on the distributed memory model. It is, however, worth noticing that the two models are not mutually exclusive. There is often much that can be gained, especially on CPUs, when the two technologies and memory models combine their strengths.

IV. GPGPU Offloading

This section first gives a brief introduction to GPGPUs, before continuing with introducing Nvidia's CUDA platform. The section then summarizes Suda *et al.*'s [10] evaluation of the GPGPU (through CUDA) performance, comparing it with University of Tokyo's supercomputer T2k. It continues by summarize Satish *et al.*'s [11] design, implementation, and evaluation of sorting algorithms for Manycore GPUs. Finally, the section finishes with summarizing Newburn *et al.*'s report describing the utilization and evaluation of Intel's Xeon Phi^{TM}.

A. GPGPUs

The use of additional processing units has been a part of computer history, for as long as computers have had a history. In the beginning processing units were highly specialized, but over time the CPU emerged and started to handle a wide array of different applications/programs.

In the early years of "external" processing units connected to the motherboard containing the CPU, were more often than not Graphic Processing Units (GPUs). This was because of the heavy load graphical applications put on the CPU and thereby slowed the whole system down.

With Moore's law making the construction of more complex GPUs cheaper, and the discovery of better graphical algorithms, GPUs rose as the forefront example of accelerators available to more than the select few, very rich, companies.

In later years, since the mid 2000s, GPUs have been promoted as General Purpose GPUs (GPGPUs), which with their efficient and cost-effective vector-operations can perform many more concurrent tasks concurrently than a CPU, given that the tasks performed are very similar to one another.

With regards to the "Brick Wall", GPGPUs avoid the "Power/Heat Wall" by specializing in vector instructions, hiding their latencies behind the number of operations they can perform in a clock cycle. The "ILP Wall" is easily avoided, by the vectorization implicitly requiring that the programs that run efficiently on GPGPUs to exploit "flat data parallelism" to gain speedups over regular CPUs. Finally, the "Memory Wall", which poses the biggest "threat" to GPGPU's speedups over regular CPUs, is mainly overcome by optimizing the CPU-GPU transfers through techniques such as *memory coalescing*, in addition to the programs requiring sufficient computations per data-element, which can be executed faster on the vectorized GPGPU, to hide the memory latencies.

B. CUDA Intro

CUDA, an acronym for *Compute Unified Device Architecture*, is an API and a parallel computing platform utilized on Nvidia GPUs, created by Nvidia. CUDA works as a language extension to the programming languages C, C++, and Fortran, and can with its own compiler generate programs runnable on Nvidia's GPGPUs.

CUDA was the first widely used GPGPU platform on the market and replaced advanced API solutions like Direct3D and OpenGL. The CUDA platform also gives support for OpenACC and OpenCL.

CUDA utilizes and relies on the *Single Program Multiple Data* paradigm, meaning that there is one program which utilizes the GPGPU for several smaller *kernels* within the program, which are intended to run multiple executions of instructions from subsections of the programs algorithm on the GPU on multiple elements of data.

The CUDA GPGPUs work by having a radically different processing architecture than typical CPUs. Instead of having deep pipelines, lots of caches, prefetching, and Out-of-Order execution, Nvidia's GPU architectures focuses on vectorizing lots of threads to hide their lack of the aforementioned CPU hardware optimizations.

Each GPGPU supports a certain number of *Streaming Multiprocessors* (SMs). Each SM supports the concurrent execution of a max number of threads, divided by blocks. In Nvidia, threads perform instructions in lockstep by multiples of 32, with 32 threads being called a "warp". The invocation of a CUDA kernel to be executed on an Nvidia GPGPU is prefixed by a number of blocks (and potentially their distribution across up to three dimensions), and amount of threads (also potentially distributed across up to three dimensions).

Thus, each invocation of a CUDA kernel will be scheduled to run as many of the available blocks concurrently as possible, with the scheduling executing any remaining blocks as resources become available.

In CUDA, especially in later architecture generations of Nvidia's GPGPUs, several different types of memory are available to the kernels executed on the GPU. While each SM has registers that their threads can use, instead of having caches, they have a large, slow global memory, a faster, read-only constant memory, and a small read/write shared memory, which all the threads in an SM share.

Thus, the kernels with the higher throughputs written in CUDA utilize all layers of memory according to their strengths, use memory coalescing with the vectorized threads, and diminish branching within warps, so as to avoid divergent threads in CUDA, among other optimizations.

Suda *et al.* [10] realizes that the "Power (Heat) Wall" poses a limitation on supercomputer's design, they experiment with GPUs that have a much better power-performance ratio than conventional CPUs.

They note that while one of the bigger challenges of achieving speedups through the use of GPUs is the latency of CPU-GPU data transfers, the performance metrics between GPUs and the T2K supercomputer at the University of Tokyo are not different by an order of magnitude.

Suda *et al.* [10] point out the following four main differences between GPUs and current supercomputers:

- SIMD vector length: Of which the GPU has a lot more
- Memory Size: Of which the GPU has a lot less compared to systems of RAM and CPUs.
- Absence of low latency cache: Which the GPU has none of.
- Register Spill Penalty: Which is much higher on the GPU due to their memory hierarchy. This difference is also exacerbated by the lack of low latency caches for the threads in the GPU.

C. CUDA Sorting

Satish *et al.* [11] report a design of high-performance parallel sort routines for manycore GPUs through CUDA. They specifically focus on two classes of sorting algorithms: radix sort and merge sort. They report up to 4x speedup on radix sort compared to the graphics-based GPUSort, and 3.5x speedup compared to comparable routines on an 8-core 2.33GHz Intel Core2 Xeon system.

Their mergesort also performs favorably, with roughly 2x speedup over GPUSort, and they report it being similarly faster than the CUDPP radix sort.

They achieved these speedups through careful exposure of substantial fine-grained parallelism and decomposing independent computiational tasks that perform minimal communication. They report that they also had to utilize the shared memory available on the GPU smartly to achieve their results, even with the synchronization costs incurred.

Their results collected were run on a Nvidia GeForce 8800 Ultra, running on a PC with 2.13 GHZ Intel Core2 6400 CPU, 2GB main memory, and using a Linux 2.6.9 kernel OS.

D. Intel's Xeon Phi

While Intel was not the first on the market with GPGPUs, they have in recent times focused heavily on the Intel Xeon Phi$^{\text{TM}}$in the HPC market.

Newburn *et al.* [4] evaluate and describe the implementation of benchmarks utilizing the Intel Xeon Phi with several different calculation heavy benchmarks. They get their speedups by comparing the runtimes with running the benchmarks on the host CPU alone.

While the transfer of memory to and from externally connected processing units such as GPGPUs on the PCI-Express bus is how the "Memory Wall" makes itself known to GPGPUs, the paper of Newburn *et al.* [4] shows that if the effort is made to write programs that implement the use of Xeon Phis, especially those that may be calculation heavy like Seismic, Astronomic, Physics, or Financial applications, can benefit from the use of a Xeon Phi.

They report results from 1.4x to 6.92x with offloading to the Xeon Phi when compared to running without, on a 2.6GHz Xeon E5-2670 Crown Pass platform running RHEL 6.2, kernel 2.6.32 64bit OS, with Pre-Production Intel Xeon Phi$^{\text{TM}}$with 61 4-thread cores running at 1.09GHz, with 8GB memory.

The Xeon Phi does stand apart from the typical GPGPU, such as those of Nvidia and AMD, by virtue of being able to be utilized in the compilation of code, as well as not following the typical threads/warps/blocks/grids/SM architecture as Nvidia does.

However, much like CUDA and OpenCL, the advantages of Intel's Xeon Phi heavily rely on the programmer utilizing the tools made available through the compiler.

V. Cross-platform/Heterogeneous Parallel Programming

This section first introduces OpenCL, before comparing it quickly with CUDA and quickly introducing LLVM, before summarizing Magni *et al.*'s [12] thread-coarsening optimization.

A. OpenCL

While the majority of systems in the history of parallel programming and HPC have been homogeneous systems, systems like personal computer's CPUs and attached GPUs make a heterogeneous system by their combined nature.

As such, the versatility of heterogeneous systems is much greater than homogeneous systems, but the efforts and difficulty necessary for their utilization can also be very great.

Open Computing Language (OpenCL)$^{\text{TM}}$is an open standard for parallel programming of heterogeneous systems, created to alleviate the difficulty of effectively utilizing heterogeneous systems. This does not mean that OpenCL cannot be utilized to write programs for homogeneous systems/platforms, but in such cases it is often more efficient to write efficient code in the programming environment/language intended for said platform.

OpenCL works in the fashion that code written in the C/C++ API and Kernel language is not compiled when written as most programs are. Rather, the compilation is performed at runtime, so as to be compiled to fit the *Instruction Set Architecture* (ISA) on which it is executed.

The Khronos group developing OpenCL state that OpenCL is meant to target supercomputers, embedded systems, mobile devices, and that one "code tree" can be executed on CPUs, GPUs, *Dynamic Signal Processors* (DSPs), *Field Programmable Gate Arrays* (FPGAs), and hardware.

As such, it requires kernels written as in CUDA to perform the computational tasks one wants to run on hardware besides the Host CPU executing the code but being much more verbose than CUDA, it permits much more complicated systems with its API to decide on what systems the code should run, if available.

While writing programs in OpenCL greatly reduces the overhead of writing code per platform/language/hardware, writing optimal/optimized code still requires manual platform/hardware specific optimizations. As such, many GPGPU developers choose to remain with the less verbose, and more easy to program CUDA platform, when writing GPGPU code.

OpenCL is thus in many ways considered an alternative to CUDA, but an alternative that as opposed to CUDA that supports heterogeneous/cross-platform end-targets, and which CUDA supports.

So while OpenCL offers and supports more varied, and perhaps more versatile parallel programming,

B. LLVM

LLVM (former *Low-Level Virtual Machine*) is a compiler infrastructure designed to be a set of re-usable libraries with well-defined interfaces.

While it has some support for better ease-of-use and general usability than GCC, while retaining full compatibility with GCCs compilation flags and commands, it has not yet matured to produce code as equally fast as GCC can.

Thus, while it is often used by compilers supporting a multitude of languages during development of code, it is then often discarded for GCC when producing the release version of the product, whenever GCC supports the language developed in.

Being a back-end compiler, LLVM was originally written to be a replacement for the existing code generator in the GCC stack. LLVM currently supports C/C++, ADA, D, Delphi, Fortran, Objective-C, OpenGL Shading Language, Go, Haskell, Java Bytecode, Julia, Swift, Python Ruby, Rust Scala, C# and more using various front ends.

Among the most famous front end compilers using LLVM is Clang, a new compiler supporting C/C++ and Objective-C. Clang is very popular for its much more human-readable error messages while still retaining full support of the language standards GCC supports.

C. Thread-Coarsening

Magni *et al.* [12] report on a compiler transformation called *thread-coarsening*, and evaluate its effects across a range of devices using OpenCL compiled through LLVM.

They evaluate the transformation on 17 benchmarks and five platforms using a multitude of different parameters, achieving speedups over 9x on individual applications and average speedups ranging from 1.15x on the Nvidia Kepler GPU to 1.5x on the AMD Cypress GPU.

They implement their benchmarks in OpenCL, and use LLVM as a source-to-source compiler, testing different thread-coarsening parameters during the cross-compilation.

The idea of thread-coarsening is that instead of having each thread only do one thing per kernel call on GPGPUs as traditionally done, they have the threads perform the same operation, such as matrix transposition, on more than one data element per thread, all the while reducing the total number of threads. The reduction of the total number of threads also frees up more resources per thread, which helps in achieving the speedups gained by this transformation.

They also compare their GPGPU results from AMD Radeon HD 5900, AMD Tahiti 7970, Nvidia GTX 480, and Nvidia K20c with a Intel Core i7-3820 using a Linux kernel 3.1.10 for all but the 480, which uses 3.2.0 for OS.

VI. Summary

This survey report presents several different platforms, technologies, libraries/APIs that enable and are utilized in parallel programming, on both homogeneous and heterogeneous systems/architectures.

The architectures/systems range from single-core, multi-thread processors, to supercomputers with many concurrently running cores, each concurrently running many threads, to GPGPUs running hundreds of concurrent threads in conjunction with aforementioned multitudes of cores and threads.

Where there is no "one size fits all" solutions, there are many good ones that fit each their own niche. If there's anything the reader should take with himself/herself from this survey of such broad topics as necessitated by the requirements of TDT24, it's that there are three major hindrances discussed in literature preventing the continuation of Moore's Law regarding computing power. And that there are many different and interesting attempts to conquer these hindrances, many of which rely on parallelism in one form or another.

And it's a selection of the most prominent, and recent techniques and technologies this report attempts to introduce to the reader, showing that while the "cheap and easy" path of exponential improvement at constant cost is over, there is still attainable room for improving computing power.

References

[1] G. E. Moore, "Readings in Computer Architecture," M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. Cramming More Components Onto Integrated Circuits, pp. 56–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=333067.333074

[2] J. Manferdelli, N. Govindaraju, and C. Crall, "Challenges and opportunities in many-core computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 808–815, May 2008.

[3] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, "Autotuning sparse matrix-vector multiplication for multicore," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-215, Nov 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-215.html

[4] C. J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire, "Offload Compiler Runtime for the Intel&#174; Xeon Phi&#153; Coprocessor," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1213–1225. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2013.251

[5] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[6] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[7] J. L. Gustafson, "Reevaluating Amdahl's Law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988. [Online]. Available: http://doi.acm.org/10.1145/42411.42415

[8] X.-H. Sun and L. M. Ni, "Another View on Parallel Speedup," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '90. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 324–333. [Online]. Available: http://dl.acm.org/citation.cfm?id=110382.110450

[9] X.-H. Sun and Y. Chen, "Reevaluating amdahl's law in the multicore era," *Journal of Parallel and Distributed Computing*, vol. 70, no. 2, pp. 183 – 188, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731509000884

[10] R. Suda, T. Aoki, S. Hirasawa, A. Nukada, H. Honda, and S. Matsuoka, "Aspects of GPU for General Purpose High Performance Computing," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 216–223. [Online]. Available: http://dl.acm.org/citation.cfm?id=1509633.1509696

[11] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161005

[12] A. Magni, C. Dubach, and M. F. P. O'Boyle, "A Large-scale Cross-architecture Evaluation of Thread-coarsening," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503268