A new computing architecture for Big Data and AI applications

Introduction

Most compute technologists acknowledge that the world's data is doubling every two years. There is also an expectation that, by 2020, datasets will exceed 21 zettabytes or more for those being used in advanced applications like derivatives of artificial intelligence (AI), such as machine learning, robotics, autonomous driving, and analytics, as well as financial markets.

These types of applications are all reliant on Big Data, which requires real-time computing of extremely small, but heavily layered, software algorithms that are ill-suited for today's multi-core processing technologies and architectures, both in terms of performance and efficiency.

The Problem: "Where did all the performance go?"

Until the early 2000s, computer performance was tied to the clock frequency of the processor, Moore's Law allowing the doubling of frequency every 18 months. Post 2005, Moore's Law allowed the doubling of processor cores every 18 months. Conventional thinking was that advances in compute performance would then come from adding additional cores to a traditional processor. However, the resulting outcome was one of diminishing returns as illustrated in Figure 1.

![Figure 1. Multi-core CPU performance is not keeping up (Click Here to see a larger image. Source: CORNAMI)](image)

The more cores added to boost performance of a traditional software algorithm, the smaller the incremental gain (with some notable exceptions). Thus, the core counts stagnated. Rather than adding more cores, the doubling of the number of transistors each generation was instead used to make the "few" cores marginally faster. The exponential increases in transistor resources are wasted, used to only give us small linear gains in performance as illustrated in Figure 2.
The amount of data that needs to be processed is increasing at an exponential rate. The current generation of the "cloud" is made up of oceans of racks filled with "few core" servers. The result is a plethora of networked servers that’s creating massive data-center sprawl, all of which leads to significant increases in power usage and carbon footprint throughout the world.

Critically, an inspection of the workloads of Big Data and Machine Learning today show that they are vastly different than the workloads existing processors were designed and optimized over decades to handle. Big Data/Machine Learning code size is measured in units of kLOC (thousand lines of code) versus the mLOC (million lines of code) units associated with traditional software (think of your favorite office suite or even your favorite operating system). For example, a simple LOC grep on the popular [BigDataBench 3.2](https://example.com) from the Chinese Academy of Sciences for the SPARK applications framework, covering a dozen different benchmarks shows a total cumulative kLOC of under 1000 lines of SCALA codes. The Yahoo! Streaming Benchmark, less than 300 lines of SCALA code. Google's TensorFlow MNIST, the entirety of the tutorial with multiple copies, a shade over 1000 lines of python code.

Key observations are as follows:

1. The workloads we need to process in Big Data/ML are uniquely different, individually small and the performance comes from replicating this code across many, many servers.
2. The amount of data that needs to be processed is increasing exponentially yearly, streaming in nature, and with some real-time requirements (think autonomous cars or timely mobile advertisements).
3. Thanks to the R&D of the silicon vendors, we still have exponential increases in transistor counts...
4. ...Yet are we getting only linear increases in compute performance generation after generation of processor.

The premise: A new processing architecture suited to today's workloads that offers a scalable, massively parallel, sea-of-cores approach is required. Let's use our transistor budgets to provide more computation by adding a massive number of cores rather than trying to speed up a few cores.

CORNAMI, an AI high-performance computing company in Silicon Valley, is addressing these issues in processing and taking compute performance to extraordinary levels. CORNAMI has developed and patented a new computing architecture using concurrency technology that uniquely changes software performance and reduces power usage, latency, and platform footprint.

The result is a massively parallel architecture with independent decision-making capabilities at each processing core, interspersed with high-speed memory, and all interconnected by a biologically-inspired network to produce a scalable sea-of-cores. This architecture is based on a unique fabric developed by CORNAMI, called TruStream Compute Fabric (TSCF), which is extensible across multiple chips, boards, and racks, with each core being independently programmable.
By using the TruStream Programming Model (TSPM), multi-core processor resources are abstracted into a common homogenous core pool. TruStream is implemented in both in software and hardware and runs across the TruStream Compute Fabric. Programmers can easily implement concurrency through CORNAMI's TruStream control structures that are embedded in higher-level standard languages.

The Heart of a Machine Learning Processing Engine
Recent announcements from Microsoft about its Convolution Neural Network Accelerator, from Google about its Tensor Processing Unit (TPU) for neural network acceleration, and from NVIDIA on their Volta for Deep Learning GPU, all reveal a similar approach to accelerating machine learning algorithms.

The algorithms to be accelerated are all variations of a three-dimensional filter (Figure 3). The heavy lifter, the algorithm that consumes the largest proportion of CPU/GPU cycles or Silicon Area, is a 3D Convolutional Filter -- this is the heart and soul of Convolution Neural Nets (CNNs).

Figure 3. Convolutional Neural Network (CNN) showing 3D Convolution + 3D ReLu + 3D Pooling (Image reprinted under the creative commons attribution license from: Gao, F.; Huang, T.; Wang, J.; Sun, J.; Hussain, A.; Yang, E. Dual-Branch Deep Convolution Neural Network for Polarimetric SAR Image Classification. Appl. Sci. 2017, 7, 447.)

To accelerate these algorithms, a two-dimensional silicon structure -- a silicon accelerator, known as a systolic array\(^1\) is used (\(^1\)"Systolic arrays for (VLSI)\", H.T. Kung and Charles E. Leiserson, 1978). Figure 4 shows two simple 3x3 matrixes that are to be multiplied together, while Figure 5 shows the systolic array of 3x3 Multiply Accumulate Units that will perform the multiplications.
For example, the output matrix element C11 contains the result $A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$ after the A and B input data has finished streaming in. Larger dimensional arrays allow a massive amount of parallelism to occur; more intricate interconnections among array elements increase the bandwidth; more sophisticated functions per element allow both convolution and other functions to be performed; and more elaborate control circuitry allow the results to be streamed out, as more data streams in and more computations are performed.

These are the techniques used are used in production parts. Strip away the complexities and the core idea is simple. Note that the input matrixes are streamed in a specific order into the left and top of the arrays with the results streaming out of the array. The attributes of a systolic array -- namely the ability to stream large amounts of data into and out of the structure continuously, simultaneous operation of all the elements, and the fact that intermediate values are kept internal to the systolic array and do not need to be "saved" to memory -- provide the required performance.
Based on the above, we can now derive a couple of key insights as follows:

- Each element of the systolic array is a form of a cellular automata that reacts to the data around it using simple rules.
- The systolic arrays in the high-performance Machine Learning Silicon Parts from Microsoft, Google, and NVIDIA (as discussed earlier in this article) treat the array as a fixed-function silicon accelerator in which the dimensions, element functionality, and interconnection were fixed when the ASIC mask set was created.
- The rapid-fire pace of announcements of new silicon accelerators for Machine Learning shows that the algorithms are still in a high degree of change. New algorithms with alternative, more efficient approaches to processing are being published almost weekly. Committing a machine learning algorithm to silicon dooms it to rapid obsolescent with next week's announcements.

The premise: What if we allow the programmer, to define the dimensionality, functionality, and interconnectivity of any systolic array via software? That is, keep the high performance that Machine Learning requires, which is a side-effect of the systolic arrays, yet allow the software programmer to keep pace with the new algorithmic developments by allowing changes entirely in software.

The additional benefit is that this software approach allows applicability to a large class of problems, well beyond just Machine Learning, in which operations are performed on elements -- cells, pixels, voxels -- and their neighbors in a two-or-higher-dimensional array. Examples of these sorts of problems are as follows:

- Machine learning
- Image processing
- Compression/Decompression
- Encoding/Decoding
- Three-dimensional (3D) modeling
- Fluid dynamics
- Finite element analysis
- Cryptography
- Error correction coding
- Modeling physical reality

The Simplest Systolic Array -- The Game of Life
Let us walk through a complete example of the aforementioned software defined systolic array approach using the TruStream Programming Model (TSPM) for software execution on a sea-of-cores TruStream Compute Fabric (TSCF). To illustrate the techniques involved in applying TruStreams to these problems, we've chosen the Game-of-Life cellular automaton, which consists of a two-dimensional grid of cells, each interacting with only its nearest neighbors. This should be familiar to almost all software programmers and simple enough to illustrate the mechanics. When the TSPM program for the Game of Life runs on the TSCF, each Game-of-Life cell runs on its own unique TSCF core.

The TruStream Programming Model
The problem of assembling systolic arrays is very much akin to the problem factory architects have been dealing with since... well, since the days of Henry Ford. Factory architects see the problem as one of organizing a collection of entities -- machines, robots, human workers -- that interact through streams of widgets flowing from one entity to another. When these architects draw diagrams of their factories, they draw block diagrams, not flow charts. There is, nevertheless, a role for flow charts in architecting factories: They are perfectly suited to describing the sequential behavior of individual machines, robots, and human workers. To summarize:
Determining the sequence of tasks a worker is to perform and laying out a factory are two fundamentally different activities.

It is these observations that inspire our view of computing: A computer is a factory for assembling data values, and inspire our view of programming: Some portions of a program are most easily and naturally expressed as flow charts; Some portions of a program are most easily and naturally expressed as block diagrams. Or equivalently: Some portions of a program are most easily and naturally expressed in the TSPM thread domain; Some portions of a program are most easily and naturally expressed in the TSPM stream domain.

This brings us to the TruStream Programming Model, which is based upon five C++ classes as follows:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>threadModule</td>
<td>Encapsulates thread-domain code:</td>
</tr>
<tr>
<td></td>
<td>-- Sequential code with TrueStream gets and puts.</td>
</tr>
<tr>
<td></td>
<td>-- NO parallelism is expressed here.</td>
</tr>
<tr>
<td>inputStream&lt;T&gt;</td>
<td>A threadModule input.</td>
</tr>
<tr>
<td>outputStream&lt;T&gt;</td>
<td>A threadModule output.</td>
</tr>
<tr>
<td>streamModule</td>
<td>Encapsulates stream-domain code:</td>
</tr>
<tr>
<td></td>
<td>-- Defines a TruStream topology (block diagram).</td>
</tr>
<tr>
<td></td>
<td>-- ALL parallelism is expressed here.</td>
</tr>
<tr>
<td>stream&lt;T&gt;</td>
<td>A TruStream</td>
</tr>
<tr>
<td></td>
<td>-- Connects one or more threadModule outputs to one or more threadModule inputs.</td>
</tr>
<tr>
<td></td>
<td>-- Appears only inside a streamModule.</td>
</tr>
</tbody>
</table>

Figure 6. The C++ classes that make up the TSPM for C++ (Source: CORNAMI)

To gain an intuitive understanding of these classes, refer to Figures 7 and 8, which provide graphical representations of the five types of TruStream objects.
Figure 7 illustrates two key features of the TruStream Programming Model as follows:

- TruStream topologies may be cyclic (or acyclic).
- Hierarchical topologies are created by nesting streamModules.
The TruStream Programming Model has three distinctive properties as follows:

- TSPM code is partitioned into two domains: the thread domain and the stream domain.
- The TSPM expresses parallelism exclusively in the stream domain.
- The TSPM relies on a single mechanism for synchronizing threads: TruStreams.

The first property, which embodies the separation of concerns design principle, allows programmers to focus exclusively on a separate concern within each domain.

- In the case of the thread domain, a programmer deals with a strictly sequential problem.
- In the case of the stream domain, a programmer deals with streams and transformations on streams.

As a result of this separation of concerns, neither the thread-domain programmer nor the stream-domain programmer is worried about multithreading or mapping code to individual processor cores.

The second property means that the thread-domain programmer -- the programmer who is writing sequential code -- does not have to deal with parallelism. This is a remarkable departure from conventional approaches to parallel programming.

The third property means that neither the thread-domain programmer nor the stream-domain programmer has to deal with the synchronization issues arising from thread-based parallelism constructs (see Figure 10). However, despite the absence of these constructs, TruStream programmers have access to all forms of parallelism.

In the Game-of-Life example below, the entirety of the TSPM program consists of definitions of four C++ classes as illustrated in Figure 9:

![Figure 9. The classes that comprise the game of life (Source: CORNAMI)](image)

![Figure 10. Constructs NOT in the TruStream Programming Model (Source: CORNAMI)](image)
The TruStream Compute Fabric

The TruStream Compute Fabric is a scalable sea-of-core processor containing both processing cores and on-die SRAM. All the cores are independently programmable and can make individual decisions in parallel -- in contrast to those of modern-day graphics processing units -- and the fabric can be extended across multiple chips, boards, and racks. Just as most programmers do not overly concern themselves with the amount of RAM they use, TruStream programmers do not have to overly concern themselves with the number of cores they use.

The connections between the TSCF and TSPM are contained in the following three properties:

- Each threadModule in a TruStream program gets its own TruStream-Compute-Fabric core.
- The only code running in a TruStream-Compute-Fabric core is the thread-domain code encapsulated in a threadModule.
- Each streamModule in a TruStream program configures a TruStream topology in the TruStream Compute Fabric.

The key features of the TruStream Compute Fabric are summarized as follows:

- The TSCF is optimized for the TruStream Programming Model, and so the TSCF -- like the TSPM -- supports all forms of parallelism.
- The TSCF has hardware support for TruStreams and for TruStream gets and puts, making the TSCF extraordinarily efficient at running TSPM programs.
- The only code running in the TSCF is application code! That's it. There is none of the bloat code listed in 11 running in the fabric.
- Threads in the TruStream Compute Fabric wait for input data and output space. Aside from that, threads do not wait.
- Each TSCF core supports -- at most -- one thread, and so there is no context switching. That means that there is no need for schedulers or dispatchers.
- TSCF performance comes from parallelism, not brute-force speed, and therefore we have the luxury of optimizing the TSCF for energy efficiency.
- Because going off-die is a performance killer, the TSCF provides a large amount of on-die SRAM.
- Each TSCF core has its own memory port, and so the TSCF delivers extraordinary aggregate memory bandwidth.

Taken together, these features lead to (1) a significant reduction in latency, (2) a significant increase in performance, and (3) a significant increase in compute power.
The Game-of-Life Cellular Automaton in TruStreams

The *Game of Life*, also known simply as *Life*, is a cellular automaton devised by the British mathematician John Horton Conway in 1970.

The "game" is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves or -- for advanced "players" -- by creating patterns with particular properties.

**Rules:**
The *universe of the Game of Life* is an infinite two-dimensional orthogonal grid of square *cells*, each of which is in one of two possible states, *alive* or *dead* (or "populated" or "unpopulated"). Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step-in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if caused by under population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

To this definition, we make two simple changes as follows:

- Instead of an infinite two-dimensional orthogonal grid of square cells, the TruStream program below implements an 8X8 grid of cells.
- The north-south and west-east edges of the array wrap around so that when the classic *Game-of-Life glider* reaches the southeast corner of the array, it disintegrates, only to reassemble in the northwest corner of the array.

The TruStream program for the Game of Life, which is described in the following sections, assumes the #includes and #defines shown in Listing 1.

```
#include <stdio.h>
#include "trustreams.hpp"
#define nNEIGHBORS 8 // Number of cell neighbors
#define nROWS 8 // Number of rows in cell array
#define nCOLS 8 // Number of columns in cell array
#define nGENERATIONS 10000 // Number of GOL "generations"
```

Listing 1. #includes and #defines (Source: CORNAMI)

**The Cell Thread Module**
The *cell* thread module class shown in Listing 2 implements the Game-of-Life *cell* described above, including the four *transition* rules.
Listing 2. The `cell` thread module (Source: CORNAMI)

Click Here to see discussions and observations regarding the `cell` thread module.

The GameOfLifeArray Stream Module
The `GameOfLifeArray` stream module (Listing 3) constructs the Game-of-Life array:
Listing 3. The `GameOfLifeArray` stream module (Source: CORNAMI)

Click Here to see discussions and observations regarding the `GameOfLifeArray` stream module.

**The Display Thread Module**

The display thread module (Listing 4) prints the cell states of successive generations of the `GameOfLifeArray` stream module.
The class display: public threadModule
{
    inputStream<bool> inStrm[nROWS][nCOLS];   // Input streams
    void code()   // Thread-domain code
    {
        int g, r, c;
        bool cellState;
        for (g = 0; g < nGENERATIONS; ++g) // Print generations
        {
            printf("\nGENERATION %d", g);
            for (r = 0; r < nROWS; ++r)
            {
                printf("\n");
                for (c = 0; c < nCOLS; ++c)
                {
                    inStrm[r][c] >> cellState; // Get cell state
                    printf("%s", ((cellState == 0) ? " - " :
                        " X ");
                }
                printf("\n");
            }
            printf("\nDone\n\n");
            endTopology(0);
        }
    }
};

Listing 4. The display thread module (Source: CORNAMI)

Click Here to see discussions and observations regarding the display thread module.

The GameOfLife Stream Module
The GameOfLife stream module (Listing 5) connects the GameOfLifeArray to the display to construct the Game of Life.

class GameOfLife: public streamModule
{
    GameOfLifeArray GOLA;   // GameOfLife object
display       D;   // display object

public:
    GameOfLife()   // Constructor
    {
        GOLA >> D;    // Connect outputs of GOLA to inputs of D
        end();        // Housekeeping
    }
};

Listing 5. The GameOfLife stream module (Source: CORNAMI)

Click Here to see discussions and observations regarding the GameOfLife stream module.

The Main Function
We now come to the simplest part of our Game-of-Life program: running it.

Listing 6. The `main()` function (Source: CORNAMI)

As illustrated in Listing 6, running a TruStream program consists of three steps:

1. Construct either a `threadModule` object (not very interesting) or a `streamModule` object (very interesting). In our example, that is accomplished by the statement:
   ```
   GameOfLife GOL;
   ```

2. Call the module member function `run()` for the object. In our example, this is accomplished by the statement:
   ```
   GOL.run();
   ```
   This call causes 10,000 generations of the Game-of-Life program to be run on 65 cores of the TruStream Compute Fabric -- 64 cores for the GOL array and one core for the display. In the Game-of-Life output (Figure 12), we see the classic Game-of-Life glider (a) gliding in a southeasterly direction in Generations 0 to 16, (b) disintegrating in Generations 17 to 26, (c) reassembling in Generation 27, and (d) returning to its initial state in Generation 32.

3. Call the module member function `wait()` for the object. In our example, this is accomplished by the statement:
   ```
   GOL.wait();
   ```
   This call causes the `main()` function to `wait` until the program completes.

After Step 3 is completed, `main()` returns. The first 33 generations of output are shown below.

```
<table>
<thead>
<tr>
<th>GENERATION 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>- - - - - - -</td>
</tr>
<tr>
<td>- - X - - - -</td>
</tr>
<tr>
<td>- - x x x - -</td>
</tr>
<tr>
<td>- X X X - - -</td>
</tr>
<tr>
<td>- - - - - - -</td>
</tr>
<tr>
<td>- - - - - - -</td>
</tr>
<tr>
<td>- - - - - - -</td>
</tr>
<tr>
<td>- - - - - - -</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GENERATION 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>- - - - - - -</td>
</tr>
<tr>
<td>- - - - - - -</td>
</tr>
<tr>
<td>- x x x - - -</td>
</tr>
<tr>
<td>- - x x x - -</td>
</tr>
<tr>
<td>- - x - - - -</td>
</tr>
<tr>
<td>- - - - - - -</td>
</tr>
<tr>
<td>- - - - - - -</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GENERATION 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>- - - - - - -</td>
</tr>
</tbody>
</table>
```
GENERATION 3
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

GENERATION 4
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

GENERATION 5
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

GENERATION 6
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

GENERATION 7
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

GENERATION 8
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
Conclusion
TruStream technology redefines what it means to compute. Gone is the thread-centric view that has dominated computing for the last 70 years. In its place is a model in which threads still play a role -- but a supporting role (it is possible to eliminate threads entirely; for example, by replacing sequential thread-domain code with functional code written in a purely functional language, and by replacing von-Neumann cores with stack machines).

The most distinctive features of the technology are (a) the separation of code into two domains: the thread domain and the stream domain, (b) the expression of parallelism exclusively in the stream domain, and (c) the reliance on a single mechanism for synchronizing threads: TruStreams.

These features have many benefits, chief among them being the following:

- The ability to rapidly, in software, construct arbitrary multidimensional systolic arrays and have them execute on a sea-of-cores computation fabric. You can implement today's favorite machine learning algorithm, and when a better one comes along, change it. When the system workload changes, the software can change the size and nature of the systolic arrays in real time. Need to perform complex and arbitrary control functions in conjunction with the machine learning function? Code it up! We cannot stress the advantages of software flexibility enough. With a high-performance silicon accelerator for machine learning, you are locked into the design decisions that were made when that ASIC was taped out, and -- to add insult to injury -- the silicon accelerators still need access to general purpose processor cores for control decisions.

- An arbitrarily scalable multi-core architecture. Most multi-core approaches have a centralized resource -- such as a scheduler or dispatcher -- that acts as a bottleneck limiting system size. In the TruStream Compute Fabric, there are no such bottlenecks. Once a TruStream topology is downloaded to the TSCF and begins running, each thread module in the topology is on its own. It gets data values from input streams, performs computations, and puts data values into output streams. That's it. There is no runtime scheduling or dispatching, and there are no synchronization primitives -- beyond TruStreams. Furthermore, because each thread module gets its own core, there is no context switching. Multiple ICs can join together to build larger Compute Fabrics; multiple servers in a rack can join together; and multiple racks can join together to process larger workloads.

- For Big Data and Machine Learning applications, the advantage shifts to having more, and more, smaller cores rather than trying to marginally increase the performance of a large, traditional, processor cores. A technology node change typically doubles the number of transistors that you can place on the same sized die. What this means in practice in today's technology nodes is that you are given another few billion transistors to play with. A traditional approach would take these transistors and translate them into a 15% to 30% performance gain. Now, you can translate them into an additional 4,000 processor cores.

Paul Master is co-founder and CTO at CORNAMI, Inc.; Frederick Furtek is co-founder and Chief Scientist at CORNAMI, Inc.