Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation

Mihaela Maliţa
St. Anselm College
Manchester, NH, USA
mmalita@anselm.edu

Gheorghe Ştefan
BrightScale Inc.
Sunnyvale, CA, USA
gstefan@brightscale.com

Dominique Thiébaut
Smith College
Northampton, MA, USA
thiebaut@cs.smith.edu

ABSTRACT

Recent embedded systems have switched to fully programmable parallel architectures. To make sure all corner cases usually present in real applications are supported and efficiently implemented in this switch of implementation, new solutions must be found. We introduce the *integral parallel architecture* (IPA) as a solution supporting intensive data computation in System-on-a-chip (Soc) implementations, fitting in a *small area*, and requiring *low power*. An IPA supports naturally all three possible styles of parallelism: *data*, *time*, and *speculative*.

As an illustrative example, we present the ***BA1024*** chip, a fully programmable SoC designed by *BrightScale, Inc.* for HDTV codecs. Its main performance figures include $60\,GOPS/Watt$ and $2\,GOPS/mm^2$, representing an efficient IPA approach for embedded computation.

Categories and Subject Descriptors

C.1.4 [**Parallel Architectures**]: Miscellaneous; C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and Embedded Systems

Keywords

Parallel architectures, embedded systems, programmable systems, video processing

1. INTRODUCTION

Technology is currently going through some important evolutionary trends that have been identified by several researchers, including Borkar [2], and Asanovic [1]. One trend is the slow-down of the increasing rate of the clock speed. Another one is the switch from pure standard functionality to more specific functionality in video, graphics, and performance hungry applications requiring full programmability. A third one is the replacement of *Application Specific Integrated Circuits* (ASIC) by programmable *Systems on a Chip* (Soc) due to development costs and increasing technological difficulties associated with the former.

Borkar and Asanovic propose several new approaches for computer architectures to respond to these changes and curb their effects: application domain oriented architectures in two versions: **many**- or **multi**-processors, or computation type oriented architectures. Intel presents a good example of the first type of architecture in its *Recognition, Mining and Synthesis* (RMS) white paper [2], while Asanovic provides an example of the second architecture in [1].

In this paper we propose two solutions to address some of the limitations imposed by the current technology shifts. The first is an optimized approach for *low-power and small-area* embedded computation in SoC. The second is a way to remove some limitations categorized by Asanovic [1] as the 13th Dwarf, and qualified as an "automaton-style" computation.

The validity of the solutions we present here rests on two hypotheses. One is that programmable SoCs can compete with ASICs only if a fully programmable parallel architecture is used, because a circuit is an intrinsically parallel system. The second hypothesis holds that the computational model of partial recursive functions [4] must be able to treat equally well both circuits, and parallel programmable systems.

Our approach naturally leads to two main results:

- the definition of an IPA[1] for intensive computations in embedded systems, and
- the proposal of a more nuanced taxonomy of parallel computation as opposed to the more structural and functional approach first introduced by Flynn [3].

Both results are exemplified in the *BA1024* [7, 8, 9] which we believe is the first embodiment of an IPA. The BA1024 is initially targeted to the HDTV market, but because of its fully programmability can support other applications [11, 10, 5].

Parallel computation is becoming ever more ubiquitous, and

[1]The reader is invited to see our approach as being different from the *heterogeneous computing systems* which are those with a range of diverse computing resources that can be local to one another or geographically distributed.

manifests itself in two extreme forms, one in *complex* computation and the other in *intense* data-parallel computation. Our paper deals with the second form. The remainder of this paper is structured as follows. In Section 2 we introduce different views and trends of the parallel computing landscape as seen by researchers at Intel and at Berkeley. In Section 3 we present partial recursiveness and show how it can be used to identify different parallel constructs. Section 4 presents a new taxonomy of computing, contrasting it with Flynn's. This leads us in Section 5 to the definition of integral parallel architectures, and the two types of forms in which they appear. Section 6 presents a state of the art circuit for video decoding and shows how it embodies the different types of parallelism we have presented. Section 7 concludes with pertinent remarks about current design trends, and how theyh can achieve the performance levels required by tomorrow's applications.

2. INTEL'S RMS AND BERKLEY'S DWARF APPROACHES

Intel's approach makes the distinction between *multi*-core era and *many*-core era, between scalar and parallel applications, and massively parallel applications. For embedded systems a many-core approach seems to be the solution because we must substitute SoC implemented in ASIC technologies with SoC implemented as fully programmable solutions. In this case, the functional granularity in ASICs must have a counterpart in the granularity of the parallel programmable system used in high-performance embedded applications.

Many-core approaches also present low-power solutions where simple processing elements (PEs) are fully utilized in each clock cycle to perform either simple functions, one per clock cycle, or complex functions composed of simpler ones.

In his paper describing the landscape of parallel computing research as seen from Berkeley [1], Asanovic makes the following statement: *We argue that these two ends of the computing spectrum* [embedded and high performance computing] *have more in common looking forward than they did in the past. First, both are concerned with power, ... Second, both are concerned with hardware utilization, ... Third, ... the importance of software reuse must increase.*

The architecture we propose here is mainly oriented to solve the first two issues, *low-power and hardware reuse*, using many small and simple programmable, though not reconfigurable elements. To define this architecture, we start from the computational model which has a very direct circuit or computing network implementation: the *partial recursive function* model, which we present in the next section.

3. FROM PARTIAL RECURSIVENESS TO PARALLEL COMPUTATION

We claim that the most suggestive classic computational model for defining parallel architectures is the model of partial recursive functions [4], because the rules defining it have a direct correspondence with circuits, the intrinsic parallel support for computation.

Composition and basic parallel structures

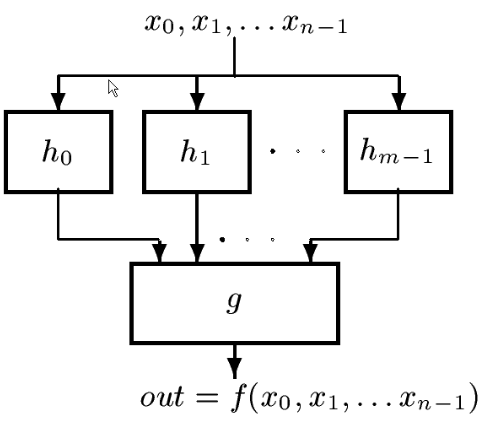

Figure 1: The physical structure associated to the composition rule.
The composition of the function g with the functions $h_0, \dots, h_{m-1}$ implies a two-level system. The first level, performing in **parallel** m computations is **serially** connected with the second level which performs a reduction function.

The first rule of composition provides the basic parallel structures to be used in defining all the forms of parallelism.

Assume we have m n-ary functions $h_i(x_0, \dots x_{n-1})$, for $i = 0, 1, \dots m-1$, along with an m-ary function $g(y_0, \dots y_{m-1})$. We can define the composition rule by combining them as follows: $f(x_0, \dots x_{n-1}) = g(h_0(x_0, \dots x_{n-1}), \dots h_{m-1}(x_0, \dots x_{n-1}))$. The physical structure associated with this concept, containing simple circuits or simple programmable machines is illustrated in Figure 1.

This suggests the following four separate and meaningful forms of parallelism, as illustrated in Figure 2:

1. **data parallel composition**: when $n = m$, each function $h_i = h$ depends on a single input variable x_i, for $i = 0, 1, \dots n-1$, and g performs the identity function (see Figure 2a). Given an input **vector** containing n scalars:
$$\mathbf{X} = \{x_0, x_1, \dots, x_{n-1}\}$$
the result is another vector:
$$\{h(x_0), h(x_1), \dots, h(x_{n-1})\}$$
2. **speculative composition**: when $n = 1$, i.e. $x_0 = x$, (see Figure 2b), g performs the identity function. In other words, it computes a vector of functions: $\mathbf{H} = [h_0, \dots h_{m-1}]$ on the same **scalar** input x, generating a vector of results:
$$\mathbf{H}(x) = \{h_0(x), h_1(x), \dots, h_{m-1}(x)\}$$
3. **serial composition**: This corresponds to the case $n = m = 1$, as shown in Figure 2c. Here, a *pipe* of different machines receives a **stream** of n scalars as input:
$$< \mathbf{X} > = < x_0, x_1, \dots, x_{n-1} >$$

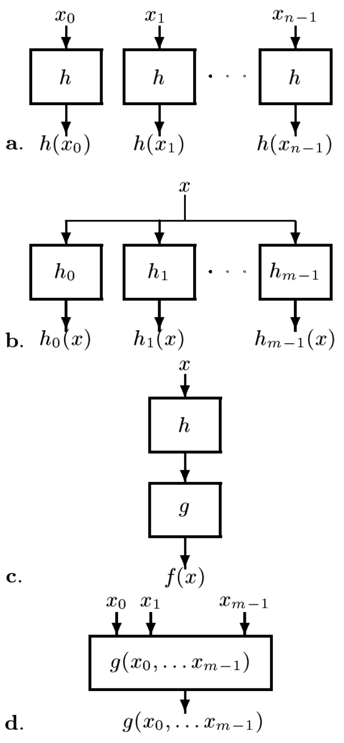

Figure 2: The four simple forms of composition. **a.** Data parallel composition. **b.** Speculative composition. **c.** Serial composition. **d.** Reduction composition.

and provides another stream of scalars

$$< f(x_0), f(x_1), \ldots, f(x_{n-1}) >$$

In the general case, the function $f(x)$ is a composition of more than two functions h and g. Thus, the function f can be expressed as a vector of functions $\mathbf{F}$ receiving as input a data stream $< \mathbf{X} >$: $\mathbf{F}(< \mathbf{X} >) = [f_0(x), \ldots f_{p-1}(x)]$. In Figure 2c we have $\mathbf{F}(< X >) = [h(x), g(x)]$

4. **reduction composition**: In this last case, each h_i performs the identity function, as is illustrated in Figure 2d. The input is the vector $\{x_0, \ldots, x_{m-1}\}$ to the block in which the function g transforms the stream of vectors into a stream of scalars $g(x_0, \ldots x_{m-1})$.

The composition rule provides the context for defining computation using the following basic concepts:

scalar : x

vector : $\mathbf{X} = \{x_0, x_1, \ldots, x_{n-1}\}$

stream : $< \mathbf{X} > = < x_0, x_1, \ldots, x_{n-1} >$

function : $f(x)$

vector of functions :

- $\mathbf{F}(< \mathbf{X} >) = [f_0(x), \ldots f_{p-1}(x)]$ applied on streams
- $\mathbf{F}(\mathbf{x}) = [f_0(x), \ldots f_{p-1}(x)]$ applied on scalars.

We now have all that is required to define the concepts of primitive recursive rule and of minimalization in our context.

Primitive recursive rule

The primitive recursive rule computes the function $f(x, y)$ as follows: $f(x, y) = h(x, f(x, y-1))$, where: $f(x, 0) = g(x)$. This rule can be translated in the following serial composition:

$$f(x, y) = h(x, h(x, h(x, \ldots h(x, g(x)) \ldots))).$$

There are two ways to implement in parallel the primitive recursive rule. In both cases we assume that a large amount of data is available for the computation. This information is in the form of vectors or streams of data ready for input into a primitive recursive function.

If the function $f(x, y)$ is to be computed for the vector of scalars $\{\mathbf{X}\} = \{y_0, y_1, \ldots, y_{n-1}\}$, then a data parallel structure is used. In this case each machine computes the function $f(x, y_i)$ using a local *data loop*. The resulting computation takes $max(y_0, y_1, \ldots, y_{n-1})$ cycles.

If, on the other hand, the function $f(x, y)$ is to be computed for a stream of scalars, then a time parallel structure is used. A pipe of n machine is selected. The pipe receives in each cycle a new scalar from the stream of scalars. In cases where $y > n$, then a *data loop* can be added, connecting the output of the pipe back to its input.

Minimalization

The minimalization rule assumes a function $f(x)$ defined as follows

$$f(x) = min(y) \text{ s.t. } m(x, y) = 0$$

The value of $f(x)$ is the minimum y for which $m(x, y) = 0$.

Here as well, two different parallel solutions are possible for the minimalization problem: one that uses data parallel structures and one that uses time parallel structures.

The first implementation of the minimilization function is the brute-force approach, and uses the speculative structure. This is illustrated in Figure 2b where each block computes a function which returns a pair containing a predicate and a scalar: $h_i = \{(m(x, i) = 0), i\}$, for $i = 1, 2, \ldots$, after which a reduction step (using a structure belonging to the class represented in Figure 2d) selects the smallest i from all pairs having the form $\{1, i\}$, if any. These pairs were generated on the previous speculative composition level. Note that all pairs of the form $\{0, i\}$ are ignored.

The second implementation of the minimalization operation shows up in time-parallel environments where speculation can be used to speed-up pipeline processing. **Reconfigurable pipes** can be conceived and implemented using special reduction features distributed along a pipe. We formalize this concept now.

We define a function pipe as the function vector:

$$\mathbf{P} = [f_0(x), \ldots f_{p-1}(x)]$$

where $y_i = f_i(x)$, for $i = 0, \ldots p-1$. The associated reconfigurable pipe transforms the original pipe characterized by $\mathbf{P} = [\ldots f_i(y_{i-1}), \ldots]$ into a pipe characterized by: $\mathbf{P} = [\ldots f_i(y_{i-1}, \ldots y_{i-s}), \ldots]$ where: $f_i(y_{i-1}, \ldots y_{i-s})$ is a function or a program that decides in each step, or cycle, which variable to use in the current computation, selecting[2] one of the $\{y_{i-1}, \ldots y_{i-s}\}$ variables with the help of an s-input selector. The maximum *degree of speculation* in this case is the index s.

Examples

We now present two typical examples of minimalization, and how speculative composition can be used to solve them and accelerate the computation.

1. Let $f(x, y)$ be a function $f : \mathbf{N}^2 \rightarrow \mathbf{N}$, where $\mathbf{N}$ denotes the set of positive integers. We need to compute all the successive positive values i and $i+1$ for $x = a$ where $sign(f(a, i)) \neq sign(f(a, i+1))$ (if any). In other words, the function finds indices associated with the different values of x where $f(x)$ has different signs for consecutive indices.

One solution involves a pool of machines performing speculative composition, as illustrated in Figure 2b, where $h_i = f(a, i)$, followed by an appropriate reduction composition, illustrated in Figure 2d. $\diamond$

2. Consider a pipeline in which we must compute the following C-style conditional expressions in one of the middle stages:

```
if ( z[n-1]==1 )
   z = z + (x-c);
else
   z = z + (x+c);
```

where `c` is a constant value, and `z[n-1]` is the most significant bit of `z`.

To maximize speed, the pipeline must evaluate $x - c$ and $x + c$ in parallel, but only one of the resulting values is fed to the next stage.

The physical structure of the pipe must be dynamically reconfigured in order to accommodate the use of a 2-PE speculative composition array, as illustrated in Figure 2b.

If the variable x is computed in Stage i of the pipe ($x = y_i = f_i(\ldots)$), then Stage $i+1$ computes $x - c = y_{i+1} = f_{i+1}(y_i)$ Stage $i+2$ computes $x + c = y_{i+2} = f_{i+2}(y_i)$ and Stage $i+3$ computes $z = y_{i+3} = f_{i+3}(y_{i+1}, y_{i+2})$. The pipe is configured as a "cross" using the stages $i+1$ and $i+2$ as a speculative bar orthogonal to the pipe. The degree of speculation in this case is $s = 2$. $\diamond$

[2]selection is one of the simplest reduction function

In order to implement speculative execution there is no need for any special features in a data-parallel environment. But for the time-parallel environment simple reduction features, or selection functions[3], must be added in each pipeline stage. This results in a *2-dimension* $n \times s$ *pipe*, i.e. an n-stage pipe where each stage is able to select its input from the output of the one of the previous s stages. This 2-dimension pipe has a maximum depth of n and the maximum degree of speculation s.

In conclusion, we observe that the minimalization rule can be implemented using all the simple parallel resources, each one the embodiment of simple forms of composition.

Next we use the different forms of composition introduced to propose a new taxonomy of parallel architectures.

4. FUNCTIONAL TAXONOMY OF PARALLEL COMPUTING

The previously identified simple forms of compositions, all summarized in Figure 2, yield a functional taxonomy of parallel computation which intersects with Flynn's original taxonomy [3], and which we categorize as follows:

data-parallel computing : this form uses operators that take vectors as arguments and returns vectors, scalars (by reduction operations) or streams (input values for time-parallel computations). This form is very similar to Flynn's SIMD machine.

time-parallel computing : this form uses operators that take streams as arguments and return streams, scalars, or vectors which can be used as input values for data-parallel computations. This form is akin to MIMD machines, but refers to the computation of a single function (described by a vector of functions) instead of multi-threading computation.

speculative-parallel computing : in this form, operators take scalars as arguments and return vectors reduced to scalars using selection. This form is used mainly to speed up time-parallel computations, and it contains a true MISD-like structure. This form has no real implementations in Flynn's taxonomy.

By positioning ourselves in the context of the partial recursive functions model, we argue that we can devise a functional taxonomy of parallel computing which better encompasses today's architectures because it uses *functions and variables* rather then instructions and data.

Since we now are in a new theoretical framework, we must define what is an architecture in this context, and also what the machines that implement this architecture are.

In the next section we present the concept of *Integral Parallel Architecture*, or IPA, as a parallel architecture featuring a multitude of the above mentioned parallel forms.

[3]This recursive function is called *projection* in the theory of partial recursive functions.

5. IPAS AND LOW-POWER EMBEDDED COMPUTATION

It today's technology ring, hi-performance and low-power embedded systems compete with ASICs. We believe that the only way to compete with the ASIC approach is to reduce the granularity of the programmable computing elements to the level of the circuits performing the application oriented functions. To this end, one must rely on small multi-processor systems, rather than large many-processor systems.

IPA: A Definition

Two opposite extreme versions of IPAs exist today:

- **complex IPA**s, with all types of parallel mechanisms tightly *interleaved.*
- **intensive IPA**s, with parallel mechanisms that are highly *separated.*

Current high-performance processors have a complex IPA, because they implement all types of parallelism in a highly integrated fashion. They are examples of the first type of IPA. Indeed, a pipelined super-scalar machine with speculative execution performs *on the same structure* – a super-speculative pipe – time-parallel computations, data-parallel computations, and speculative-parallel computations.

An intensive IPA, on the other hand, is implemented using distinct hardware support for the simplified, special cases of composition previously emphasized. Because demanding applications cannot be implemented using a single type of parallelism, intensive IPAs are the only solution for building fully programmable, low-power, hi-performance systems-on-a-chip.

We continue our exploration of the concept of Intensive IPAs by looking at their implementation.

Intensive IPAs

Actual embedded applications demand intensive data-parallel computation and/or intensive time-parallel computation, often requiring the support of speculative resources. As a result the machine implementation of an IPA must provide resources for both. We coin such a machine an **integral parallel machine**, or IPM, and observe that there are two ways of designing them:

- with a unique physical support for both data- and time-parallel computation, which we refer to it as a **monolithic IPM**, or
- as two distinct structures for the two types of parallelism, which we refer to as a **segregated IPM**.

We now briefly present each type of machine.

Monolithic integral parallel machine

The main structure of this IPM type is a linear array of processing elements (PE) that operates in two modes: as a data-parallel machine receiving instructions from a controller, or as a pipeline of PEs executing their own locally stored program [6], [10].

Note that the performance requirements of today's applications will dictate the addition of speculative parallelism to this array of PEs.

However, the simplest form of this type of IPM is a linear data-parallel array of n bidirectionally connected PEs, adorned with mechanisms allowing constant time selection (s-selectors) to allow s-degree speculative executions in pipeline mode where each PE is connected to the previous s PEs.

Segregated integral parallel machine

Segregated IPMs, on the other hand, are chosen when the cost of adding the speculative resources (the s-selectors) becomes too expensive for a very long pipe. If n, the number of stages in the pipe, is in the order of several hundreds or thousand, and if the degree of speculation s is larger than 2, the additional hardware is justified only if these very long pipes are mandated by the overall design. When it is not, a short pipe with fewer stages and a reasonably large s can be added as a separate element in the overall design. In cases where the amount of data parallelism is superior to the amount of time parallelism present, a segregated IPM is preferable.

In the video domain, for example, data parallelism dominates the computation, but the time parallelism can not be neglected or else performance is not maximized. Therefore, both types of parallelism must be supported by the hardware, and their design must account for the amount of parallelism to be exploited.

Intensive IPMs: Accelerators for Complex IPA

Once the type of IPM needed is identified for a given application, either monolithic or segregated, one important problem remains: that of designing the application interface. While the kernel of an application is usually solved by *intense* computation, the application interface usually requires *complex* computation. Therefore, a fully programmable SoC must include the following:

- a complex operating-system oriented section, built around one or a few controllers each having a complex IPA, and
- an accelerator based on an uniform array of simple PEs, each built with an intensive IPA.

The main benefit of the strong separation between complex and intense IPAs is *power saving.* Complex computation is usually power hungry, while intense computation, which is simple in nature, involves uniform, small, and intensely utilized structures.

Today's designers must also deal with "leakage current effects". Reusability of the physical structure is the main weapon to fight against this new enemy. Only simple hardware can be easily reused in a programmable environment.

"No multipliers or floating point units" become the designers' slogan, because the units' transistors can not be reused. Only simple PEs with elementary arithmetic and logic units provide the ideal environment for intensive and low power computing.

6. A CASE STUDY: BA1024

In this section we present a chip currently manufactured by *BrightScale, Inc.*, the ***BA1024***, with at its core a segregated IPM of parameters $n = 1024$, $m = 8$ and $s = 4$. The chip is dimensioned to fully support the computation required by dual HDTV codecs.

Chip's Organization

The chip is a combination of many- and multi-processor architectures. It contains an array of 1024 PEs, alongside an array of 4 MIPS working as an MIMD machine. The chip is implemented in $130nm$ standard process, runs at 200MHz, and sports the following features:

- audio and video interfaces for two HD channels,
- four MIPS processors used as: host processor, video processor, demultiplexing processor, and audio processor,
- a DDR interface sustaining data transfer rates of $3.2GB/sec$,
- an interconnection "fabric" allowing transfers of 128-bit words per clock cycle, and
- an accelerator with intensive IPA containing:
  - 1024 16-bit PEs, each having an integer unit, a boolean unit, and 256 16-bit words of data memory (the data parallel section),
  - a global loop structure, used mainly to identify the first PU having the selected predicate set to true,
  - a reduction tree, used to extract data from the array, or to calculate the number of PEs with a predicate set to true,
  - a bidirectional I/O tree, which transfers data to and from the array (the transfer is strictly parallel and concurrent with the main computational process),
  - 8 16-bit PEs, each with its own program memory and implementing the time parallel section.

Because codec implementation is data intensive, the data-parallel section is very heavily used. The time-parallel section is needed to accelerate purely sequential codec modules, mainly for the H.264 video standard.

Performance

The performance measures of the BA1024 are summarized in Tables 1 and 2. Note that our goal in presenting the BA1024 data is solely to emphasize the significant differences between complex and intensive IPAs, and by no means do we claim any superiority of the BrightScale design over that of other companies. The dot product operation in Table 1 refers to vectors having 1024 16-bit integer components. Note also that the 200 GOPS figure does not include floating point operations, which are emulated in software. In this case 2 GFLOPS + 100 GOPS can be sustained. Also note that the DCT operations in Table 2 are performed on 8×8 arrays.

Operations	Performance
16-bit operations	200 GOPS
Dot product	28 clock cycles $>$ 7 GDotProducts/sec
External bandwidth	3.2 GB/sec
Internal bandwidth	400 GB/sec
OPS/power	$>$ 60 GOPS/Watt
OPS/area	$>$ 2 GOPS/mm^2

Table 1: Overall performance data for the BA1024.

Operations	Performance
DCT	0.15 cycles/pixel
SAD	0.0025 cycles/pixel
deblocking filter	0.12 cycles/pixel
decoding of H.265 dual HD stream	85% utilization of accelerator

Table 2: Video performance data for the BA1024.

7. CONCLUDING REMARKS

Ubiquitous parallelism.. Parallelism is an integral part of today's medium and high performance processors. It is exploited more and more in the two IPA variants that we have presented: complex IPAs, where parallelism appears in general purpose processors, and in intensive IPAs, such as specialized accelerators. Chances are that the term "parallel" may become obsolete in the near future, as technological limitations are removed, and all computations end up implementing parallelism in a "natural" way.

Complex versus intensive IPA.. There is a significant difference between the performance measures of standard processors using complex IPAs and those of machines making use of intensive IPAs, of which the Brightscale BA1024 is an example. Typical values for today's complex IPAs are

- 4 GIPS + 4 GFLOPS
- **(0.04 GIPS + 0.04 GFLOPS)/Watt**
- (0.016 GIPS + 0.016 GFLOPS)/mm^2

where an instruction is a 32-bit operation. For intensive IPAs, however, the measures are:

- 200 GOPS **or** 2 GFLOPS + 100 GOPS
- **60 GOPS/Watt** or
  (0.6 GFLOPS + 30 GOPS)/Watt

- 2 GOPS/mm^2.

The most important difference appears in power saving of more than 90% for the intensive computation performed by an IPA.

Segregation of the complex IPA from the intensive IPA.. Segregation of IPAs is the best solution for optimizing both *price(area) versus performance* and *power versus performance*.

High Performance Architecture=
complex IPA + intensive IPA.

Almost all demanding applications would benefit from the use of a new kind of computing architecture. We claim that in the case of such an architecture, **the complex part must be strongly segregated from the intensive part** in order to reach the target performance at a competitive price, and with a minimum amount of dissipated energy. Maximizing the intensive part area is squeezed and power is saved.

8. ACKNOWLEDGMENTS

We would like to thank Emanuele Altieri, Lazar Bivolarski, Frank Ho, Bogdan Mîţu, Tom Thomson, and Dan Tomescu for insightful discussions on the material presented here.

9. REFERENCES

[1] K. Asanovic. The landscape of parallel computing research: A view from berkeley. Technical report, U.C. Berkeley, December 2006.

[2] V. Borkar. Platform 2015: Intel processor and platform evolution for the next decade. Intel Corporation white paper, 2005.

[3] M. J. Flynn. Some computer organization and their affectiveness. *IEEE Trans. Comp.*, C21(9):948–960, September 1972.

[4] S. C. Kleene. General recursive functions of natural numbers. *Journal of Symbolic Logic*, 2, 1937.

[5] M. Malita, G. Stefan, and M. Stoian. Complex vs. intensive in parallel computation. In *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C*. ICCGI 2006, August 2006.

[6] B. Mitu. private communication, 2005.

[7] G. Stefan. The ca1024: A massively parallel processor for cost-effective hdtv. *SPRING PROCESSOR FORUM: Power-Efficient Design*, June 2006.

[8] G. Stefan. The ca1024: Soc with integral parallel architecture for hdtv processing. In *4th International System-on-Chip (SoC) Conference and Exhibit*. SOC Conference, November 2006.

[9] G. Stefan, A. Sheel, B. Mitu, T. Thomson, and D. Tomescu. The ca1024: A fully programable system-on-chip for cost-effective hdtv media processing. In *Hot Chips: A Symposium on High Performance Chips*. Stanford U., August 2006.

[10] D. Thiebaut and M. Malita. Pipelining the connex array. In *BARC 07*. BARC, January 2007.

[11] D. Thiebaut, G. Stefan, and M. Malita. Dna search and the connex technology. In *International Multi-Conference on Computing in the Global Information Technology - Challenges for the Next Generation of IT&C*. ICCGI 2006, August 2006.