Programming Constructs for Exascale Computing in Support of Space Situational Awareness

Mark Schmalz **William Chapman** **Eric Hayden**
Sanjay Ranka **Sartaj Sahni** **Gerhard Ritter**

Department of Computer and Information Science and Engineering
University of Florida, Gainesville FL 32611-6120

ABSTRACT

Increasing data burdens associated with image and signal processing in support of space situational awareness implies much-needed growth of computational throughput beyond petascale (10^{15} FLOP/s) to exascale regimes (10^{18} FLOP/s, 10^{18} bytes of memory, 10^{18} disks and Input/Output (I/O) channels, etc.) In addition to growth in applications data burden and diversity, the breadth and diversity of high performance computing architectures and their various organizations have confounded the development of a single, unifying, practicable model of parallel computation. Therefore, models for parallel exascale processing have leveraged architectural and structural idiosyncrasies, yielding potential misapplications. In response to this challenge, we have developed a concise, efficient computational paradigm and software called Program Compliant Exascale Mapping (PCEM) to facilitate efficient optimal or near-optimal mapping of annotated application codes to parallel exascale processors.

Our theory, algorithms, software, and experimental results support annotation-based parallelization of application codes for envisioned exascale architectures, based on Image Algebra (IA) [Rit01]. Because of the rigor, completeness, conciseness, and layered design of image algebra notation, application-to-architecture mapping is feasible and scalable at exascales. In particular, parallel operations and program partitions are categorized in terms of six types of parallel operations, where each type is mapped to heterogeneous exascale processors via simple rules in the PCEM annotation language.

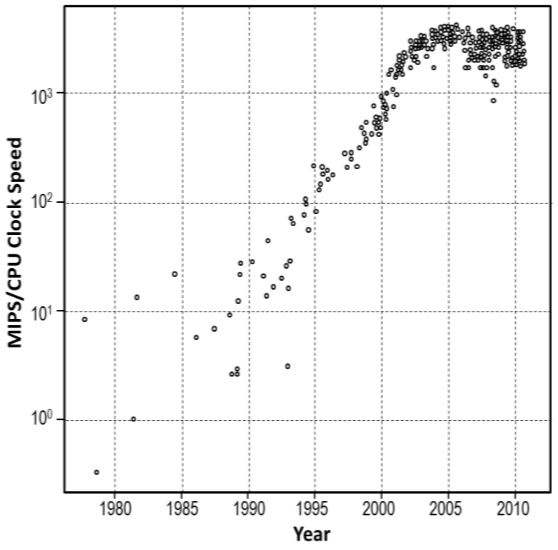
In this paper, we overview opportunities and challenges of exascale computing for image and signal processing in support of radar imaging in space situational awareness applications. We discuss software interfaces and several demonstration applications, with performance analysis and results in terms of execution time as well as memory access latencies and energy consumption for bus-connected and/or networked architectures. The feasibility of the PCEM paradigm is demonstrated by addressing four principal challenges: (1) architectural/structural diversity, parallelism, and locality, (2) masking of I/O and memory latencies, (3) scalability, and (4) efficient representation/expression of parallel applications. Examples will demonstrate how PCEM helps solve these challenges efficiently on real-world computing systems.

Keywords: High-performance computing, Exascale processing, Image and signal processing

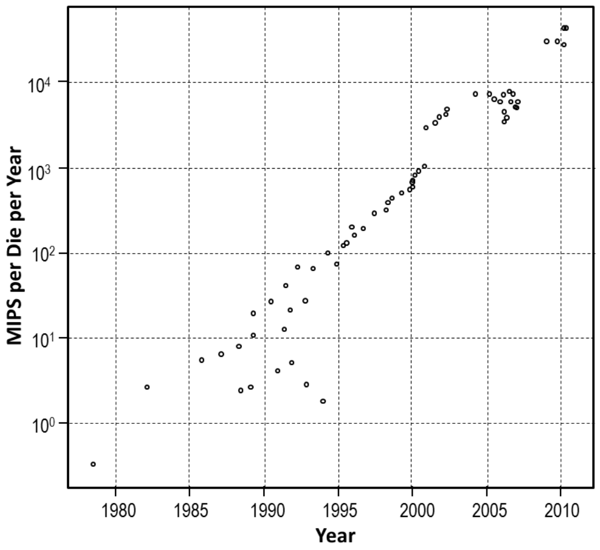
1. INTRODUCTION

Continuing increases in the number and complexity of high-performance computing applications imply an ongoing expansion of high-performance computing (HPC) capacity and throughput while maintaining or improving existing levels of numerical quality and system reliability. For several years, continued HPC throughput increases have been difficult to achieve by merely increasing processor clock rate. In particular, the power dissipation of CPU circuit technology is limited by frequency ($P = \mathbf{O}(f^2)$), capacitance and voltage ($P = \mathbf{O}(CV^2)$). Thus, the mere decrease in feature size to current dimensions (22nm [Int12]) and beyond will likely not facilitate significant increases in clock rate similar to the trends of the past four decades. This trend is illustrated graphically in Figure 1, which shows that peak clock rate has maintained at approximately 3.4GHz for several years while throughput has increased.

Several years ago, a 3.4GHz clock rate seemed to be a hard limit, as the transformation of capacitive losses to heat-transferring resistance losses at that frequency became insurmountable with commercial heat sinks. However, this engineering challenge did not stop progressive throughput increases, as shown in Figure 1b. A general observation that can be inferred from these engineering trends is that further increases in processing power are more likely to occur as a result of spatial parallelism. A leading example of this emerging trend is the graphics processing unit (GPU), which has many processing elements (PEs) or cores per die. Another example is the hybrid multiprocessor (HMP) such as Intel's *Ivy Bridge* or *Haswell* architectures [Int12].



(a)



(b)

Figure 1. Trends in CPU clock rate: (a) millions of instructions per second (MIPS) and CPU clock rate, and (b) throughput in MIPS per die, per calendar year, 1976-2011; after [Gill11].

Thus, from a statistical and mathematical computing perspective, GPU- and HMP-based HPC is the emerging norm for parallelizeable programs, functions, or operations. GPUs have, and HMPs are expected to gain, a huge installed base: it is estimated that graphics chip shipments in 2016 would approach 688 million units [Ped12]. Although centered on photorealistic video rendering for computing and the video gaming industry, such a large consumer base is advantageous to HPC. Primarily, the amortization of GPU development cost has thus far reduced unit cost to several hundred dollars for GPU chips with throughput currently ranging from 1 TFLOPs to 10 TFLOPs per chip (depending on numerical quality).

GPUs were adopted by the HPC community due to massive parallelism and (in a few cases) the ability to perform double-precision floating point computation. Scientific and military applications have benefitted from GPUs for HPC tasks, but mainly to the extent that the GPU can be employed as a co-processor. As Amdahl's Law predicts, programs with high sequential content (such as decision trees) tend not to benefit from parallel computation on a GPU or cluster of GPUs [Amd67,Cass12]. Indeed, we have found (in prior unpublished research) that some sequential applications actually run slower on a GPU than on a single-core CPU. So the HPC community tends to employ GPU-based HPC systems for natively parallel applications, for example, synthetic aperture radar (SAR) image reconstruction [Chap11], computational fluid dynamics (CFD), and finite element analysis (FEA).

As shown in Figure 2, CPUs and GPUs can be clustered – so we have determined that parallelism can be partitioned and clustered hierarchically at levels of *chip*, *device*, *subsystem, cluster*, and *super-cluster* (a cluster of clusters). It is thus reasonable to envision the extension of GPU based computing from today's devices with hundreds of cores, to current state-of-the-art petascale assemblies of GPUs comprising tens to hundreds of thousands of cores, to exascale systems comprised of super-clusters of CPU-controlled clusters of GPUs as well as petascale machines. Hence, this study emphasizes the use of GPU-based exascale systems to optimize and implement inherently parallel applications, on a production basis.

We achieve nearly-optimal mapping of image and signal processing operations to exascale architectures by exploiting parallelism inherent in *image algebra* [Rit01], a rigorous concise notation that unifies linear and nonlinear mathematics in the image domain. The image algebra research project at University of Florida determined that six types of parallel operations comprise parallel image and signal processing [Rit01,Sch10]. Our technology called *Programmable Computing with Exascale Mappings* (PCEM) maps these six operation types to CPUs and GPUs, with significant increases in computational throughput and maintenance of computational accuracy.

This paper summarizes our research via the following organization. Section 2 presents background and theory, while Section 3 discusses our implementational approach in detail. Section 4 has practical examples pertaining to image reconstruction, with conclusions and future work presented in Section 5.

2. BACKGROUND AND THEORY

We begin with an overview of parallel architectures for exascale computing (Section 2.1) and previous or related work in mapping image and signal processing operations to exascale architectures (Section 2.2). We then illustrate how image algebra is structured, and how the constituent types of parallel operations are organized for PCEM-based mapping to exascale architectures (Section 2.3).

2.1. Overview of Exascale Architecture

Let us envision an exascale processor as an heterogeneous processing unit (HPU). In this vision of HPU computing evolving within the computer science and engineering community, an exascale system could be comprised at a high level of *superclusters* connected by a *wide-area network* (WAN). For example, a supercluster could include one or more hundreds-of-petaflops machines (in a larger view) at a national laboratory. At the supercluster level, each supercluster node could be associated with (or be comprised of) a plurality of nodes at the *cluster level*.

Continuing with our vision-in-the-large, each cluster node could be associated with or comprised of *control subsystems* such as multicore CPUs, *storage* subsystems such as electro-mechanical or solid-state disks, *datapath* subsystems such as multicore CPUs and GPUs or clusters of CPU-GPU nodes, and *interface* subsystems such as sensor controllers, I/O processors and so forth. Cluster nodes could be formed by connecting these subsystems via a high-speed *local network* (LAN) such as Infiniband^{TM} [Lul11].

At a more detailed level, each subsystem node could contain one or more *devices*, each comprised of *components*. For example, within a datapath subsystem, *device level* CPUs and GPUs would contain storage, control, datapath, and communication (bus) elements defined at the *component level* of the exascale hierarchy. This view is depicted in Figure 2, which notionally illustrates our vision for PCEM-controlled exascale systems.

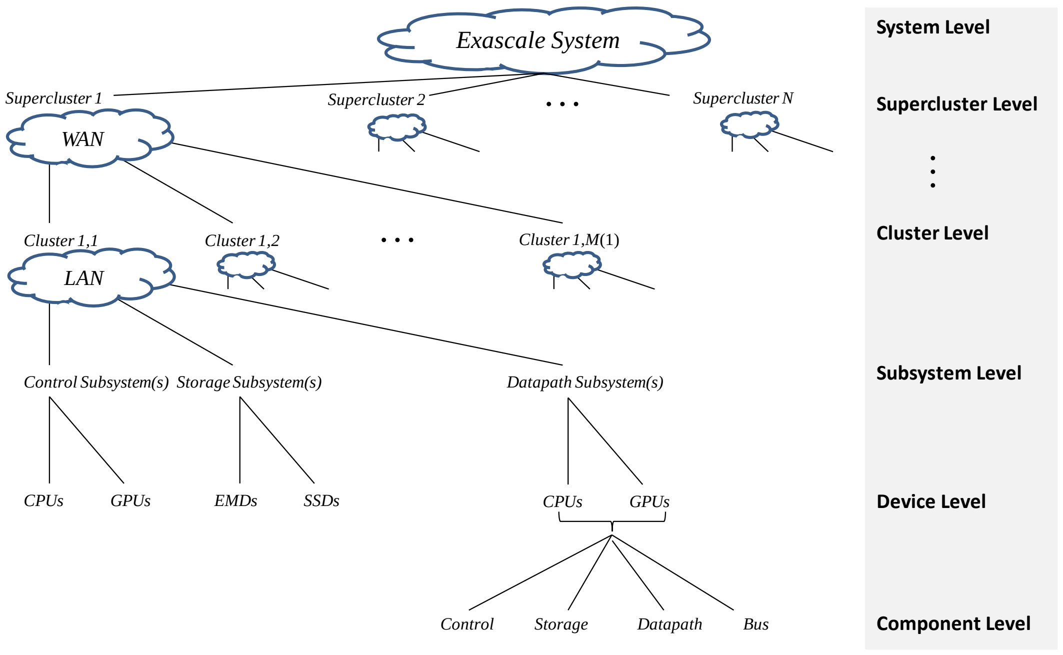


Figure 2. Notional view of an envisioned *PCEM* exascale computing hierarchy.

We present this idealized, hierarchical vision of an exascale system for two reasons. Firstly, a "holy grail" of high-performance computing is optimal mapping of work items in an application to processing elements in an architecture. This algorithm-to-architecture mapping is performed under control of an optimizing scheduler. In large computer systems such as the envisioned exascale hierarchy shown in Figure 2, each level of the system could exploit a potentially different scheduling mechanism. For example, at the supercluster level a large-task, a provably correct procedure such as OmniScheduler^{TM} [Li09] would map large jobs (requiring hours of compute time) across multiple WAN-connected computing sites. At the cluster level, a knapsack-based scheduler [Swe98] would assign smaller jobs, or smaller portions of large jobs, to many LAN-connected heterogeneous processors. At the bus-connected device level, a tightly organized scheduling algorithm could map operations or portions of operations to processing partitions that could be as small as thread blocks assigned to streaming multiprocessors.

The second reason for our envisioning of a PCEM-enabled exascale computing system as an hierarchical organization is *managed versatility*. A beautiful and powerful feature of exascale computing is that, within any level, the system designer can prescribe – or the system's hierarchically scheduled network can choreograph – fine-granular mappings of computational work to components having various levels of heterogeneity. Thus, in addition to significantly increased throughput (10^{18} FLOPs) an exascale system would have the ability to reconfigure itself to produce clusters of heterogeneous network regions – each of which can tailor itself to specific computing tasks. For realism, we assume that this would be done within an efficient reconfiguration timeframe. From our previous work in reconfigurable computing [Swe98], it is well known that if a task completes within time Δt_E, and the network reconfigures for another task within time Δt_R, efficiency and utility can be realized when $\Delta t_R << \Delta t_E$.

2.2. Overview of Previous Work

The distribution of computing tasks across a multi-level network (i.e., *WAN* > *LAN* > *bus-connected*) implies the scheduling of individual as well as composite tasks, where the latter can be comprised of many grouped or coalesced operations. Given current and expected contexts of *green computing* [Cass12], it is reasonable to trade off runtime, space consumption, computational error, and power/energy consumption behaviors and constraints. We have shown that concentrating computing effort in lower-cost nodes or distributing work across faster nodes respectively reduces power or time cost [Ahm12]. Our PCEM methodology can expand this tradeoff to achieve prespecified balance among time, space, power, time-dependent energy profile, and numerical quality variables.

PCEM can support power conservation via *increased locality* in space and time by (a) using local nodes where possible to conserve communication cost, or (b) exploiting lower-power instructions at device and component levels of the hierarchy, versus scattering a task across multiple clusters. Conversely, by assigning tasks or operations to a larger collection of datapaths, *increased parallelism* can be achieved, thereby yielding faster computation via exploiting faster local latencies versus slower global latencies. This assumes, of course, that the given task can be partitioned into subtasks, and that these decimated work units are appropriate for one or more lower-level devices.

Because of connective flexibility and heterogeneity in the exascale vision of Figure 2, and due to GPU/HMP emergence as workhorse processors for graphics and HPC, many new opportunities present for GPU/HMPs [Gill11, Ped12]. Exascale systems appear to be well suited for image reconstruction, finite element analysis, and computational fluid dynamics – problems with data points potentially at exascale quantities (e.g., 10^{18} pixels, mesh or grid elements) all related by inherently parallel functionality or data structure(s) [Ped12].

In this paper, radar image reconstruction for space situational awareness employs iterative invocation of a tensor-product-like structure, whereby each element of a sensed data array potentially contributes to every pixel of a reconstructed image [Chap11]. Applications abound – reconstruction of optical telescope images and synthetic aperture radar imagery, tomographic reconstruction for medical or security applications (e.g., CAT and MRI scanning, trans-apparel body surveillance or TABS, ground- or structure-penetrating acoustic or electromagnetic sensing for cavity or object detection). Each of these applications requires huge amounts of streaming data from airborne or standoff sensors, and features highly parallel iterative kernels that are ideal for GPUs and HMPs. Due to data movement latencies arising in part from the relative low speed of PCI Express buses, transferring these huge datasets overwhelmed the computational efficiency realized through a GPU's SIMD architecture. But an HMP, with on-chip data transfers via high-speed cache can change the processing dynamic.

Unfortunately, GPUs and HMPs represent a programming challenge for software vendors and their commercial user communities. Unlike sequential computing that enjoyed the von Neumann architecture as its unifying model for over 50 years, *there is no unifying model for parallel computing*. As a result, each operation, function, procedure, and program is currently parallelized manually or is associated with a developed, proven code fragment in one or more libraries. In the former case (manual parallelization) one typically obtains slow, error-prone code that may not be provably correct. In the latter case, one encounters rigidity that can severely limit the fluency with which parallel computation can be employed in the service of science and defense applications, and the arts.

2.3. Image Algebraic Support for PCEM-Enabled Exascale Computing

In PCEM, we exploit the underlying concept derived from the mathematical model of image algebra [Rit01] that six types of operations support parallel image/signal computing: *pointwise arithmetic*, *global reduction*, *inner product*, *convolution product*, *matrix product*, and *tensor product*. Each operation type has one or more efficient mappings to a parallel architecture such as a SIMD mesh, MIMD network, or a hybrid processor (e.g., multicore CPU with a SIMD unit or a GPU with a MIMD-partitionable set of streaming multiprocessors) [Sch10]. This allows PCEM to have versatility, flexibility, and robustness since our set of operation types is small, and each of our operation-to-architecture mappings have been specifically developed for a given processor in a provably correct manner.

Importantly, PCEM has the ability to view a program as a sequence of individual tasks, or as a collection of task subsequences. This allows PCEM to group a sequence of programmatic operations into a collection of parallel tasks, and to efficiently schedule these tasks on a MIMD network such as the exascale system illustrated in Figure 2. Support for SIMD meshes and networks of heterogeneous processors has also been designed and partially implemented. PCEM scheduling is layered, modular, extensible, and maintainable as well as amenable to centralized or distributed operation. PCEM's evolving scheduler design supports increased heterogeneity at multiple levels (see Figure 2), will allow faulting or failing devices to be present in the network but circumnavigated at runtime, and will provide fault tolerance and avoidance including redundancy (as needed) and rollback.

We next discuss key implementational details of how PCEM implements exascale mappings.

3. PROGRAM COMPLIANT EXASCALE MAPPING

The PCEM study addresses key challenges of exascale computing in the following practicable ways, aiming for the objective of near-optimal parallel scheduling:

- *Parallel operations employed in DoD and DOE applications are categorized into six classes that comprise a parallel virtual machine (PVM)*, which is amenable to being mapped to a wide variety of heterogeneous target architectures. The PCEM PVM is complete, because the six types or classes of operations employed in PCEM were proven to cover operation types in image and signal processing as well as grid- and mesh-based modeling and simulation, pattern recognition (including neural networks), and other DoD/DOE applications [Rit01]. PVM's compactness ensures that runtime support for each target processor will be compact and maintainable - thus fulfilling objectives of good software engineering, and will be extensible via specialization of each operation type, as described below.
- *Each operation type can be viewed as an object-oriented class that can be specialized to yield one or more instances specific to a given application domain*. For example, the operation type *pointwise arithmetic* can be specialized to yield *pointwise addition*, *pointwise multiplication*, *pointwise comparison*, and so forth. These specializations are described extensively in [Rit01], and are further elaborated herein for PCEM.
- *Parallel operations are annotated in source code in an easily-readable user-friendly manner* that supports understanding of the PCEM-annotated code by programmers as well as system developers. This means that, unlike some annotation systems [Dur12], PCEM notation is not cryptic and does not require manual insertion of code fragments written in arcane programming languages such as OpenCL, OpenMP, or CUDA. Thus, PCEM annotation can be readily interpreted and modified by algorithm designers or programmers.
- *Annotated operations can be identified individually or coalesced, and are scheduled near-optimally* (individually or in groups), to support PCEM's achieving our user-specified goals of joint optimization of time, space, energy profile, error (numerical quality), and power, also called *STEEP* optimization. In particular, individual operations can be identified and assigned to different processor architectures in a heterogeneous system, thereby allowing each operation to be computed on the processor that is best suited to optimization objectives. Alternatively, individual operations using common operands (large vector-parallel image structures) can be collected, then assigned to a group of processors. This *coalescing* of operations and operands is important for achieving time- and power-optimal execution by reducing data movement between target (slave) processors and their controlling (master) nodes. To support maintainability in the presence of system upgrades, each PCEM runtime module can be coded for our small set of six operation types, with new hardware specifications readily included in the PCEM scheduler's knowledge base of processor attributes and performance characteristics.
- *Exascale system status can be monitored and used to drive scheduling to achieve fault tolerance* in the case of faulting or failing processors. This ensures continuity of computation, at the expense of delays incurred by process rollback and restart. We are currently developing a "snapshot" capability within the PCEM supervisory module, to incrementally record the progress of computations on each slave processor or cluster of such processors, as well as on the master processor(s). This will support periodic monitoring of each processor's status, to support rolling back a computation incrementally when a fault or exception is detected.
- *Scheduling includes performance tradeoffs* among runtime, space consumption, energy profile, power consumption and numerical quality that can be specified directly within a PCEM annotation block, in terms of a prioritized list of performance objectives such as *runtime* > *power* > *error*. Additional variables for this optimizing scheduler will include energy profile as a function of time, numerical quality or stability, degree of parallelism, locality, and spatial extent of the target processors – as supported by hardware status availability.

- *Parallel programs can be exercised via a script-like capability* programmable by system developers, to support performance verification and optimization for different groupings of heterogeneous processors, e.g., for load balancing across multiple levels of an exascale system such as that illustrated in Figure 2.
- *PCEM innovations are encapsulated in a user-friendly compiler wrapper that runs from standard operating systems, and uses commercial-off-the-shelf compilers* such as the GNU C++ compiler, OpenMP, and Nvidia's CUDA compiler, which are available publicly. Since the PCEM system is, by design, modular and supportive of plug-ins, we are able to readily and economically support new compilers into PCEM without adversely affecting functionality of other parts of the PCEM system or of the compiler.

Thus, our PCEM technical solution directly and successfully addresses key issues of (1) *annotation-driven parallelization* of legacy code; (2) *portability* of legacy code; (3) architectural *heterogeneity*; (4) *optimization of parallelism and locality* in terms of *performance metrics* such as runtime, space and power consumption, energy profile, and numerical quality via user-specified constraints stated in the annotation; (5) *optimal partitioning of code* based on separating or coalescing of individual operations or groups of operations, respectively; (6) *low cost and ease of use* via the use of COTS compilers and operating systems as plug-in modules; (7) *increased code reliability and reduced error rate* as a result of templated, easy-to-read annotation format; and (8) *support for algorithm/system co-design* via scripts that exercise algorithms over a wide range of software and hardware performance parameters and constraints.

We achieve these objectives for each operation type, as follows.

3.1. Pointwise Operations

Consider an image domain (coordinate set) **X**, which is called a *point set*, customarily a subset of $\mathbf{R}^n$, where **R** denotes the real numbers. Let a *value set* **F** denote the values of pixels in an image. Image algebra theory defines an *image* **a** as a *mapping from a point set to a value set*, so we write $\mathbf{a} : \mathbf{X} \rightarrow \mathbf{F}$. Customarily, **F** denotes a subset of the reals, but can be any set. We also write $\mathbf{a} \in \mathbf{F}^{\mathbf{X}}$, to be more concise. Let us define an image $\mathbf{a} \in \mathbf{F}^{\mathbf{X}}$ as a set

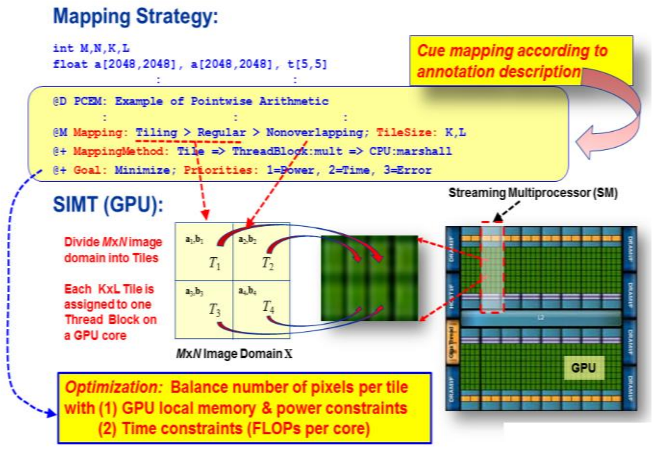
$$\mathbf{a} = \{(\mathbf{x},\mathbf{a}(\mathbf{x})) : \mathbf{a}(\mathbf{x}) \in \mathbf{F}, \mathbf{x} \in \mathbf{X}\} \,. \tag{1}$$

This allows us to view an image **a** as a collection of *pixels* where each pixel is denoted by $(\mathbf{x},\mathbf{a}(\mathbf{x}))$, with $\mathbf{a}(\mathbf{x})$ a member of the value set **F** and **x** a *point* in the point set **X**.

For pointwise operations, let the value set $\mathbf{F} = \mathbf{R}$ and let an associative, commutative function $\gamma : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$. The function γ induces a corresponding parallel operation $\gamma : \mathbf{R}^{\mathbf{X}} \times \mathbf{R}^{\mathbf{X}} \rightarrow \mathbf{R}^{\mathbf{X}}$ that can be applied to two images $\mathbf{a},\mathbf{b} \in \mathbf{R}^{\mathbf{X}}$ to yield another image $\mathbf{c} \in \mathbf{R}^{\mathbf{X}}$, defined as

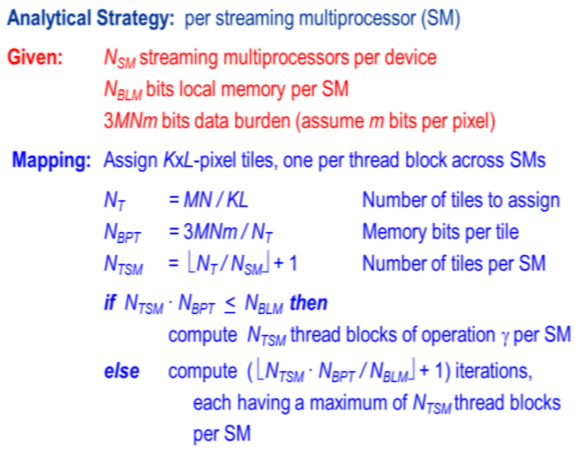
$$\mathbf{c} = \mathbf{a} \,\gamma\, \mathbf{b} = \{(\mathbf{x},\mathbf{c}(\mathbf{x})) : \mathbf{c}(\mathbf{x}) = \mathrm{a}(\mathbf{x}) \,\gamma\, \mathbf{b}(\mathbf{x}), \mathbf{x} \in \mathbf{X}\} \,. \tag{2}$$

This *pointwise* or *Hadamard* operation can be specialized. For example, if $\gamma = +$, then we have *pointwise addition*; if $\gamma = \cdot$, we have *pointwise multiplication*. Pointwise arithmetic and logic operations are defined in image algebra for any associative commutative function (e.g., +, -, x, /, $\wedge$, $\vee$). There is also a specialization of pointwise operations called pointwise unary operations – for example, $sin(\mathbf{a}) = \{(\mathbf{x},\mathbf{b}(\mathbf{x})) : \mathbf{b}(\mathbf{x}) = sin[\mathbf{a}(\mathbf{x})], \mathbf{x} \in \mathbf{X}\}$.


Mapping Strategy:
Cue mapping according to annotation description
SIMT (GPU):
Streaming Multiprocessor (SM)
Divide MxN image domain into Tiles
Each KxL Tile is assigned to one Thread Block on a GPU core
a_1,b_1
a_2,b_2
a_3,b_3
a_4,b_4
T_1
T_2
T_3
T_4
MxN Image Domain X
GPU
Optimization: Balance number of pixels per tile with (1) GPU local memory & power constraints (2) Time constraints (FLOPs per core)

```
int M,N,K,L
float a[2048,2048], a[2048,2048], t[5,5]
                  :                   :
@D PCEM: Example of Pointwise Arithmetic
        :                :              :
@M Mapping: Tiling > Regular > Nonoverlapping; TileSize: K,L
@+ MappingMethod: Tile => ThreadBlock:mult => CPU:marshall
@+ Goal: Minimize; Priorities: 1=Power, 2=Time, 3=Error
```

(a)



Analytical Strategy: per streaming multiprocessor (SM)

Given: N_{SM} streaming multiprocessors per device
N_{BLM} bits local memory per SM
$3MNm$ bits data burden (assume m bits per pixel)

Mapping: Assign KxL-pixel tiles, one per thread block across SMs

N_T	$= MN / KL$	Number of tiles to assign
N_{BPT}	$= 3MNm / N_T$	Memory bits per tile
N_{TSM}	$= \lfloor N_T / N_{SM} \rfloor + 1$	Number of tiles per SM

```
if $N_{TSM} \cdot N_{BPT} \leq N_{BLM}$ then
    compute $N_{TSM}$ thread blocks of operation $\gamma$ per SM
else
    compute $(\lfloor N_{TSM} \cdot N_{BPT} / N_{BLM} \rfloor + 1)$ iterations,
        each having a maximum of $N_{TSM}$ thread blocks per SM
```

(b)

Figure 3. Diagram of (a) mapping and scheduling, and (b) load balancing algorithm for GPU implementation of pointwise multiplication.

To map a pointwise operation to a GPU or HMP, we merely partition the source domain **X** into a set of nonoverlapping tiles U ⊂ X, then map each operand pair ($\mathbf{a}|_{\mathrm{U}}$, $\mathbf{b}|_{\mathrm{U}}$) to a streaming multiprocessor. The PCEM notation (at top of Figure 3a) specifies that **a** and **b** are 2048x2048-pixel integer-valued images and that tiling is *regular* and *nonoverlapping* with *rectangular tiles* of size *K*x*L* pixels. The mapping directive denoted by

```
Tile => ThreadBlock : mult => CPU : marshall
```
(3)

denotes that each tile is mapped to a GPU (slave) thread block which performs a pointwise multiplication operation, and the resulting tile is sent to the CPU (master processor) to be marshaled into the result image. The optimization objective is to minimize firstly *power consumption*, secondly *execution time*, and thirdly *arithmetic error*. In practice, the evolution of PCEM's annotation format has been simplified for easy insertion into the actual code where the pointwise multiplication is performed, as shown in the following format:

```
@D Example of Pointwise Arithmetic
@L Oper: Type=Pointwise; Subtype=Multiplication
@+ Precision=64;
@+ Opn1: Datatype=double; Domain=rectangular; Size1=1000; Size2=1000; Ptr=x;
@+ Opn2: Datatype=double; Domain=rectangular; Size1=1000; Size2=1000; Ptr=y;
@+ Dest: Datatype=double; Domain=rectangular; Size1=1000; Size2=1000; Ptr=z;
@M MappingType: Tiling;
@+ Goal: Minimize;
```

This annotation is processed by the PCEM interpreter and implemented as shown in Figure 3, with very little additional overhead, especially when compared with the size of the source and result images.

In the load balancing algorithm shown in Figure 3b, N_{TSM} tiles of size N_{BPT} bits per tile are assigned to each given streaming multiprocessor (SM) having local memory N_{BLM} bits. Assuming that all SMs operate in parallel, then a maximum of $\lfloor N_{TSM} \cdot N_{BPT} / N_{BLM} \rfloor$ iterations of the pointwise operation γ = · comprise the performance-limiting computational path.

3.2. Global Reduce Operations

If γ is an associative and commutative binary operation on **F** and **X** is finite, for example, X = {x_1, x_2, …, x_n}, then **γ** induces a unary operation Γ denoted abstractly as

$$\Gamma : \mathbf{F}^{\mathrm{X}} \to \mathbf{F}, \tag{4}$$

which is called the *global reduction* (or *global reduce*) *operation induced by* γ, and is defined as

$$\Gamma\mathbf{a} = \mathop{\Gamma}_{\mathbf{x}\in\mathbf{X}} \mathbf{a}(\mathbf{x}) = \mathop{\Gamma}_{k=1}^{n} \mathbf{a}(\mathbf{x}_k) = \mathbf{a}(\mathbf{x}_1)\,\gamma\,\mathbf{a}(\mathbf{x}_2)\,\gamma\cdots\gamma\,\mathbf{a}(\mathbf{x}_n) \tag{5}$$

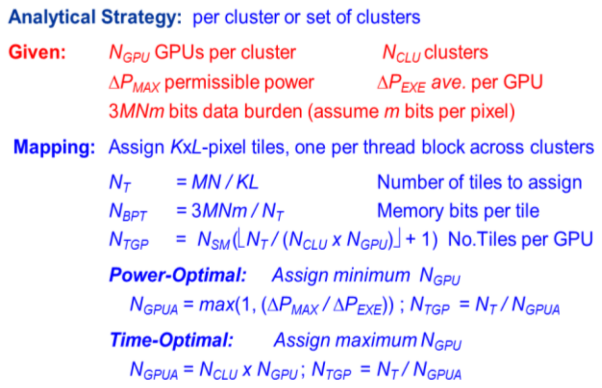
For example, if **F** = **R** and γ is the operation of addition (γ = +), then $\Gamma = \Sigma$ and

$$\Sigma\mathbf{a} = \sum_{\mathbf{x}\in\mathbf{X}} \mathbf{a}(\mathbf{x}) = \sum_{k=1}^{n} \mathbf{a}(\mathbf{x}_k) = \mathbf{a}(\mathbf{x}_1) + \mathbf{a}(\mathbf{x}_2) + \cdots + \mathbf{a}(\mathbf{x}_n) \quad . \tag{6}$$

The customary value set (**R**, ∨, ∧, +, ·) provides for four basic global reduce operations, respectively, $\bigvee\mathbf{a}$, $\bigwedge\mathbf{a}$, $\Sigma\mathbf{a}$, and $\Pi\mathbf{a}$.

```
@D Example of Global Reduction
@L Oper: Type=Reduction;
@+      Subtype=Summation
@+ Precision=64;
@+ Opn1: Datatype=double;
@+      Domain=rectangular; Size1=1000;
@+      Size2=1000; Ptr=x;
@+ Dest: Datatype=double;
@+      Domain=rectangular; Size1=1;
@+      Size2=1; Ptr=z;
@M MappingType: Tiling;
@+ Goal: Minimize;
```

(a)



Analytical Strategy: per cluster or set of clusters

Given: N_{GPU} GPUs per cluster — N_{CLU} clusters
ΔP_{MAX} permissible power — ΔP_{EXE} *ave.* per GPU
3*MNm* bits data burden (assume *m* bits per pixel)

Mapping: Assign *K*x*L*-pixel tiles, one per thread block across clusters

N_T	$= MN / KL$	Number of tiles to assign
N_{BPT}	$= 3MNm / N_T$	Memory bits per tile
N_{TGP}	$= N_{SM}(\lfloor N_T / (N_{CLU} \times N_{GPU})\rfloor + 1)$	No.Tiles per GPU

Power-Optimal: *Assign minimum* N_{GPU}
$N_{GPUA} = max(1, (\Delta P_{MAX} / \Delta P_{EXE}))$; $N_{TGP} = N_T / N_{GPUA}$

Time-Optimal: *Assign maximum* N_{GPU}
$N_{GPUA} = N_{CLU} \times N_{GPU}$; $N_{TGP} = N_T / N_{GPUA}$

(b)

Figure 4. PCEM (a) annotation and (b) load balancing algorithm for global reduction operation on a GPU.

Implementation of the global reduction operation on a GPU is an adaptation of the pointwise operation γ, from which Γ is derived. Firstly, we modify the operation type to become `GlobalReduction > Summation`, then specify the disposition of each tile as `Tile => ThreadBlock : sum => CPU : sum`.

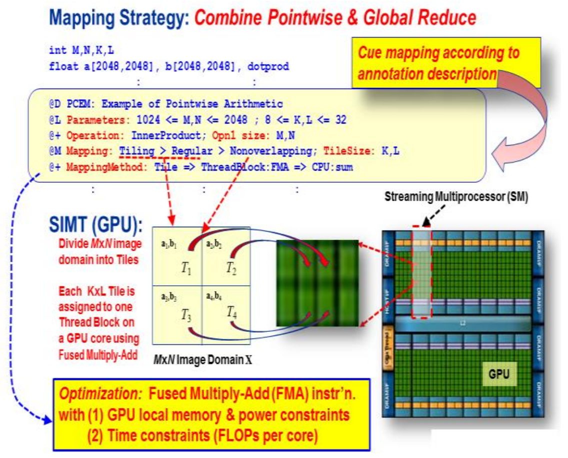
The PCEM annotation shown in Figure 4a constrains the mapping process by the same type of tiling shown in Figure 3a. The associated load balancing algorithm shown in Figure 4b is similar to the pointwise case shown in Figure 3b. As before, this code is translated to CUDA or OpenCL/MPI code with very little overhead.

3.3. Inner (Dot) Product Operation

The dot product of two vectors **a** = (a_1, a_2, ..., a_n) and **b** = (b_1, b_2, ..., b_n) is defined as:

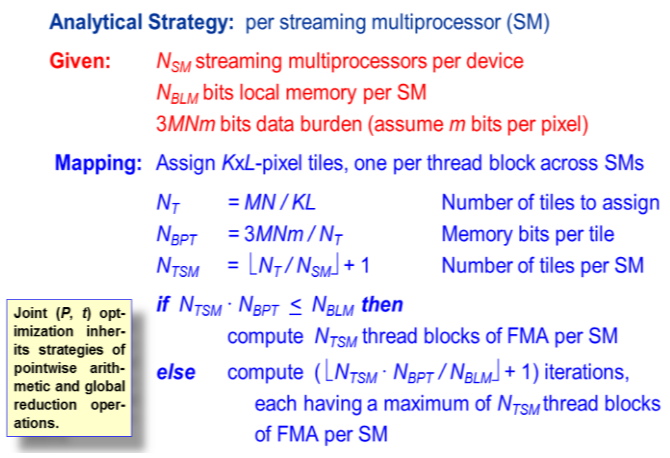
$$\mathbf{a} \bullet \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \qquad (7)$$

Given images $\mathbf{a},\mathbf{b} \in \mathbf{R}^{\mathbf{X}}$, the inner product is expressed in image algebra as $s = \mathbf{a} \bullet \mathbf{b} = \Sigma(\mathbf{a} * \mathbf{b})$. It is readily seen that the inner product is a combination of the global reduce operation of summation applied to the pointwise operation of multiplication. Thus, given the object-oriented focus of our PCEM methodology, we can apply composition to yield the mapping and load balancing strategies illustrated in Figure 5.


Mapping Strategy: Combine Pointwise & Global Reduce
Cue mapping according to annotation description
SIMT (GPU):
Divide MxN image domain into Tiles
Each KxL Tile is assigned to one Thread Block on a GPU core using Fused Multiply-Add
a_1,b_1
a_2,b_2
a_3,b_3
a_4,b_4
T_1
T_2
T_3
T_4
MxN Image Domain X
Streaming Multiprocessor (SM)
GPU
Optimization: Fused Multiply-Add (FMA) instr'n. with (1) GPU local memory & power constraints (2) Time constraints (FLOPs per core)

```
int M,N,K,L
float a[2048,2048], b[2048,2048], dotprod
        :                    :
@D PCEM: Example of Pointwise Arithmetic
@L Parameters: 1024 <= M,N <= 2048 ; 8 <= K,L <= 32
@+ Operation: InnerProduct; Opn1 size: M,N
@M Mapping: Tiling > Regular > Nonoverlapping; TileSize: K,L
@+ MappingMethod: Tile => ThreadBlock:FMA => CPU:sum
     :             :              :
```

(a)



Analytical Strategy: per streaming multiprocessor (SM)

Given: N_{SM} streaming multiprocessors per device
N_{BLM} bits local memory per SM
$3MNm$ bits data burden (assume m bits per pixel)

Mapping: Assign KxL-pixel tiles, one per thread block across SMs

N_T	$= MN / KL$	Number of tiles to assign
N_{BPT}	$= 3MNm / N_T$	Memory bits per tile
N_{TSM}	$= \lfloor N_T / N_{SM} \rfloor + 1$	Number of tiles per SM

if $N_{TSM} \cdot N_{BPT} \leq N_{BLM}$ ***then***
compute N_{TSM} thread blocks of FMA per SM
else compute $(\lfloor N_{TSM} \cdot N_{BPT} / N_{BLM} \rfloor + 1)$ iterations, each having a maximum of N_{TSM} thread blocks of FMA per SM

Joint (P, t) optimization inherits its strategies of pointwise arithmetic and global reduction operations.

(b)

Figure 5. Diagram of (a) mapping and (b) load balancing for GPU implementation of inner product.

The PCEM embedded annotation is of similar form to that for pointwise and global reduce operations:

```
@D Example of Inner Product
@L Oper: Type=InnerProduct
@+ Precision=64;
@+ Opn1: Datatype=double; Domain=rectangular; Size1=1000; Size2=1000; Ptr=x;
@+ Opn1: Datatype=double; Domain=rectangular; Size1=1000; Size2=1000; Ptr=y;
@+ Dest: Datatype=double; Domain=rectangular; Size1=1; Size2=1; Ptr=z;
@M MappingType: Tiling;
@+ Goal: Minimize;
```

Note that Figure 5 assumes the use of Nvidia's fused multiply-add (FMA) instruction, which influences our scheduling and load balancing algorithm, as FMA has the advantage of performing one multiplication and one addition operation concurrently per clock cycle, which is especially useful in the following case of convolution.

3.4. Convolution Product Operation

Consider the convolution of an image with a template. In image algebra, *templates* are images whose *values* are images [Rit01]. The concept of a template unifies and generalizes the usual concepts of templates, masks, windows, and neighborhood functions into one general mathematical entity. Also, templates generalize the notion of structuring elements used in mathematical morphology.

Definition. A template is an image whose pixel values are images (functions). An **F**-valued template from **Y** to **X** is a function $\mathbf{t} : \mathbf{Y} \rightarrow \mathbf{F}^{\mathbf{X}}$. Thus, $\mathbf{t} \in (\mathbf{F}^{\mathbf{X}})^{\mathbf{Y}}$ and **t** is an $\mathbf{F}^{\mathbf{X}}$-valued image on **Y**. For notational convenience, we define $\mathbf{t}_{\mathbf{y}} \equiv \mathbf{t}(\mathbf{y}) \;\; \forall \mathbf{y} \in \mathbf{Y}$, and call **y** the *target point* of **t**. The image $\mathbf{t}_{\mathbf{y}}$ has the following set-theoretic representation:

$$\mathbf{t}_{\mathbf{y}} = \{(\mathbf{x}, \mathbf{t}_{\mathbf{y}}(\mathbf{x})) : \mathbf{x} \in \mathbf{X}\}, \qquad (8)$$

and the pixel values $\mathbf{t}_{\mathbf{y}}(\mathbf{x})$ of this image are called the *weights* of the template $\mathbf{t}$ at point $\mathbf{y}$.

If $\mathbf{t}$ is a real- or complex-valued template from $\mathbf{Y}$ to $\mathbf{X}$, then the *support* of $\mathbf{t}$ is denoted by $\mathcal{S}(\mathbf{t}_{\mathbf{y}})$ and is defined as

$$\mathcal{S}(\mathbf{t}_{\mathbf{y}}) = \{\mathbf{x} \in \mathbf{X} : \mathbf{t}_{\mathbf{y}}(\mathbf{x}) \neq 0\}. \tag{9}$$

More generally, if $\mathbf{t} \in (\mathbf{F}^{\mathbf{X}})^{\mathbf{Y}}$ and $\mathbf{F}$ is an algebraic structure with a zero element 0, then the support of $\mathbf{t}_{\mathbf{y}}$ will be defined as $\mathcal{S}(\mathbf{t}_{\mathbf{y}}) = \{\mathbf{x} \in \mathbf{X} : \mathbf{t}_{\mathbf{y}}(\mathbf{x}) \neq 0\}$.

For extended real-valued templates we also define the following supports at infinity:

$$\mathcal{S}_{\infty}(\mathbf{t}_{\mathbf{y}}) = \{\mathbf{x} \in \mathbf{X} : \mathbf{t}_{\mathbf{y}}(\mathbf{x}) \neq \infty\} \quad \text{and} \quad \mathcal{S}_{-\infty}(\mathbf{t}_{\mathbf{y}}) = \{\mathbf{x} \in \mathbf{X} : \mathbf{t}_{\mathbf{y}}(\mathbf{x}) \neq -\infty\}. \tag{10}$$

If $\mathbf{X}$ is a space with an operation + such that (X,+) is a group, then a template $\mathbf{t} \in (\mathbf{F}^{\mathbf{X}})^{\mathbf{X}}$ is said to be *translation invariant* (with respect to the operation +) if and only if for each triple $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbf{X}$ we have that $\mathbf{t}_{\mathbf{y}}(\mathbf{x}) = \mathbf{t}_{\mathbf{y}+\mathbf{z}}(\mathbf{x}+\mathbf{z})$. Templates that are not translation invariant are called *translation variant* or, simply, *variant* templates.

The definition of an image-template product provides the rules for combining images with templates, and templates with templates. The definition of this product includes the usual correlation and convolution products employed in digital image processing. Suppose $\mathbf{F}$ is a value set with two binary operations $\bigcirc$ and γ, where $\bigcirc$ distributes over γ, and γ is associative and commutative. If $\mathbf{t} \in (\mathbf{F}^{\mathbf{X}})^{\mathbf{Y}}$, then for each $\mathbf{y} \in \mathbf{Y}$, $\mathbf{t}_{\mathbf{y}} \in \mathbf{F}^{\mathbf{X}}$. Thus, if $\mathbf{a} \in \mathbf{F}^{\mathbf{X}}$, where $\mathbf{X}$ is finite, then $\mathbf{a} \bigcirc \mathbf{t}_{\mathbf{y}} \in \mathbf{F}^{\mathbf{X}}$ and $\Gamma(\mathbf{a} \bigcirc \mathbf{t}_{\mathbf{y}}) \in \mathbf{F}$. It follows that the binary operations $\bigcirc$ and γ induce a binary operation

$$\circledast : \mathbf{F}^{\mathbf{X}} \times (\mathbf{F}^{\mathbf{X}})^{\mathbf{Y}} \to \mathbf{F}^{\mathbf{Y}} \tag{11}$$

where

$$\mathbf{b} = \mathbf{a} \circledast \mathbf{t} \in \mathbf{F}^{\mathbf{Y}} \tag{12}$$

is defined by

$$\mathbf{b}(\mathbf{y}) = \Gamma(\mathbf{a} \bigcirc \mathbf{t}_{\mathbf{y}}) = \Gamma_{\mathbf{x} \in \mathbf{X}} (\mathbf{a}(\mathbf{x}) \bigcirc \mathbf{t}_{\mathbf{y}}(\mathbf{x})). \tag{13}$$

Therefore, if $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, then

$$\mathbf{b}(\mathbf{y}) = (\mathbf{a}(\mathbf{x}_1) \bigcirc \mathbf{t}_{\mathbf{y}}(\mathbf{x}_1))\ \gamma\ (\mathbf{a}(\mathbf{x}_2) \bigcirc \mathbf{t}_{\mathbf{y}}(\mathbf{x}_2))\ \gamma \cdots \gamma\ (\mathbf{a}(\mathbf{x}_n) \bigcirc \mathbf{t}_{\mathbf{y}}(\mathbf{x}_n)). \tag{14}$$

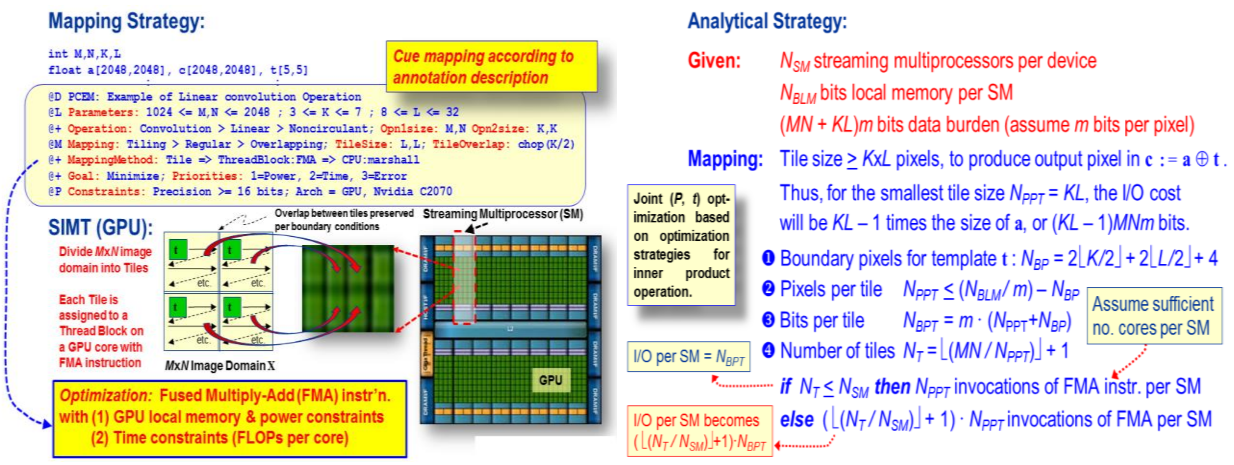
The expression $\mathbf{a} \circledast \mathbf{t}$ is called the *right convolution product of* $\mathbf{a}$ *with* $\mathbf{t}$, or more simply, the *generalized convolution product*. While $\mathbf{a}$ is an image on $\mathbf{X}$, the product $\mathbf{a} \circledast \mathbf{t}$ is an image on $\mathbf{Y}$. Thus, templates support image transformation from one type of domain to an entirely different domain type.

Replacing ($\mathbf{F}$, γ, $\bigcirc$) by ($\mathbf{R}$, +, $\cdot$) changes $\mathbf{b} = \mathbf{a} \circledast \mathbf{t}$ into $\mathbf{b} = \mathbf{a} \oplus \mathbf{t}$, which is the *linear image-template product* or, more simply, the *convolution* of $\mathbf{a}$ with $\mathbf{t}$, where

$$\mathbf{b}(\mathbf{y}) = \sum_{\mathbf{x} \in \mathbf{X}} (\mathbf{a}(\mathbf{x}) \cdot \mathbf{t}_{\mathbf{y}}(\mathbf{x})), \tag{15}$$

where $\mathbf{a} \in \mathbf{R}^{\mathbf{X}}$ and $\mathbf{t} \in (\mathbf{R}^{\mathbf{X}})^{\mathbf{Y}}$.

Figure 6 illustrates the mapping and load balancing strategies for the linear convolution operation. Observe that the convolution kernel requires overlap between applications of the template $\mathbf{t}$. This is accounted for in the load


Mapping Strategy:
Cue mapping according to annotation description
SIMT (GPU):
Divide MxN image domain into Tiles
Each Tile is assigned to a Thread Block on a GPU core with FMA instruction
Overlap between tiles preserved per boundary conditions
Streaming Multiprocessor (SM)
etc.
MxN Image Domain X
GPU
Optimization: Fused Multiply-Add (FMA) instr'n. with (1) GPU local memory & power constraints (2) Time constraints (FLOPs per core)
Analytical Strategy:
Joint (P, t) optimization based on optimization strategies for inner product operation.
I/O per SM = N_{BPT}
I/O per SM becomes $(\lfloor (N_T / N_{SM}) \rfloor + 1) \cdot N_{BPT}$
Assume sufficient no. cores per SM

```
int M,N,K,L
float a[2048,2048], c[2048,2048], t[5,5]
@D PCEM: Example of Linear convolution Operation
@L Parameters: 1024 <= M,N <= 2048 ; 3 <= K <= 7 ; 8 <= L <= 32
@+ Operation: Convolution > Linear > Noncirculant; Opn1size: M,N Opn2size: K,K
@M Mapping: Tiling > Regular > Overlapping; TileSize: L,L; TileOverlap: chop(K/2)
@+ MappingMethod: Tile => ThreadBlock:FMA => CPU:marshall
@+ Goal: Minimize; Priorities: 1=Power, 2=Time, 3=Error
@P Constraints: Precision >= 16 bits; Arch = GPU, Nvidia C2070
```

Given: N_{SM} streaming multiprocessors per device
N_{BLM} bits local memory per SM
$(MN + KL)m$ bits data burden (assume m bits per pixel)

Mapping: Tile size $\geq KxL$ pixels, to produce output pixel in $\mathbf{c} := \mathbf{a} \oplus \mathbf{t}$.
Thus, for the smallest tile size $N_{PPT} = KL$, the I/O cost will be $KL - 1$ times the size of $\mathbf{a}$, or $(KL - 1)MNm$ bits.

❶ Boundary pixels for template $\mathbf{t}$: $N_{BP} = 2\lfloor K/2 \rfloor + 2\lfloor L/2 \rfloor + 4$
❷ Pixels per tile $N_{PPT} \leq (N_{BLM} / m) - N_{BP}$
❸ Bits per tile $N_{BPT} = m \cdot (N_{PPT} + N_{BP})$
❹ Number of tiles $N_T = \lfloor (MN / N_{PPT}) \rfloor + 1$

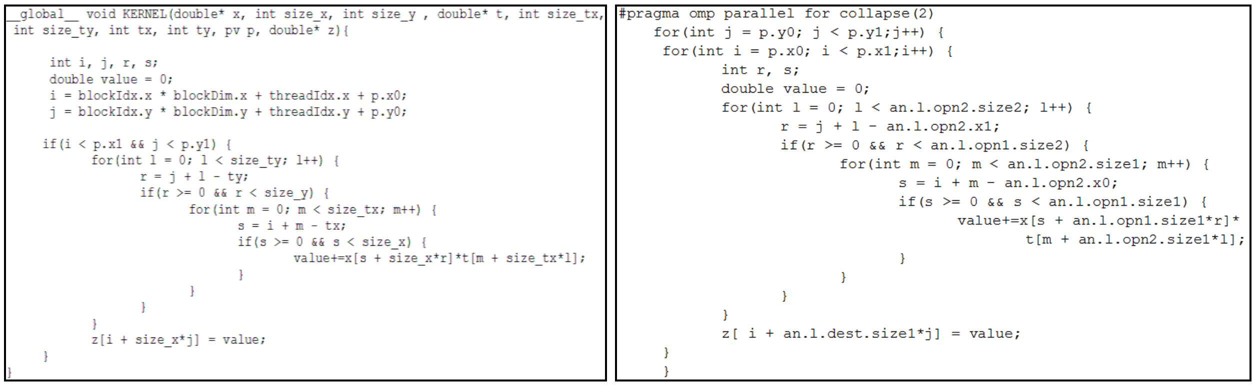
if $N_T \leq N_{SM}$ *then* N_{PPT} invocations of FMA instr. per SM
else $(\lfloor (N_T / N_{SM}) \rfloor + 1) \cdot N_{PPT}$ invocations of FMA per SM

(a) (b)

Figure 6. Notional diagram of (a) mapping and (b) load balancing for GPU implementation of convolution.

balancing strategy by including the number of boundary pixels N_{BP} in the number of pixels per tile N_{PPT}, which is used (as in the previous operation types) to calculate the number of bits per tile N_{BPT} and number of tiles N_T.

PCEM also generates code fragments, and links to compact libraries of code, for CUDA and OpenMP, as listed in Figure 7a. Here, the CUDA kernel implements linear convolution in double precision and receives pointers to *x*, *y*, and *z*, which correspond to the two inputs (image and template) and one output (image) associated with the convolution operation. A partition vector *p* contains information about which elements of the matrix this kernel processes, and the target point of template **t** is specified by (*tx*,*ty*). The variables *r* and *s* help provide a check on boundary position. The OpenMP code for linear convolution, listed in Figure 7b, uses OpenMP to perform pointwise multiplication followed by summation. The code receives an annotation structure *an* and a partition vector *p* as its inputs, then extracts pointers to operands and their dimensions from the logical structure contained within *an*. Inside each block, an OpenMP directive is used to distribute loop iterations among the available CPU cores, as illustrated in Figure 6a. Similar code generation mechanisms exist for CUDA, OpenMP, and MPI for the six types of operations listed herein.



```
__global__ void KERNEL(double* x, int size_x, int size_y , double* t, int size_tx,
 int size_ty, int tx, int ty, pv p, double* z){

     int i, j, r, s;
     double value = 0;
     i = blockIdx.x * blockDim.x + threadIdx.x + p.x0;
     j = blockIdx.y * blockDim.y + threadIdx.y + p.y0;

    if(i < p.x1 && j < p.y1) {
           for(int l = 0; l < size_ty; l++) {
                  r = j + l - ty;
                  if(r >= 0 && r < size_y) {
                         for(int m = 0; m < size_tx; m++) {
                                s = i + m - tx;
                                if(s >= 0 && s < size_x) {
                                       value+=x[s + size_x*r]*t[m + size_tx*l];
                                }
                         }
                  }
           }
           z[i + size_x*j] = value;
    }
}
```

```
#pragma omp parallel for collapse(2)
     for(int j = p.y0; j < p.y1;j++) {
      for(int i = p.x0; i < p.x1;i++) {
             int r, s;
             double value = 0;
             for(int l = 0; l < an.l.opn2.size2; l++) {
                     r = j + l - an.l.opn2.x1;
                     if(r >= 0 && r < an.l.opn1.size2) {
                             for(int m = 0; m < an.l.opn2.size1; m++) {
                                     s = i + m - an.l.opn2.x0;
                                     if(s >= 0 && s < an.l.opn1.size1) {
                                             value+=x[s + an.l.opn1.size1*r]*
                                                       t[m + an.l.opn2.size1*l];
                                     }
                             }
                     }
             }
             z[ i + an.l.dest.size1*j] = value;
     }
     }
```

(a) (b)

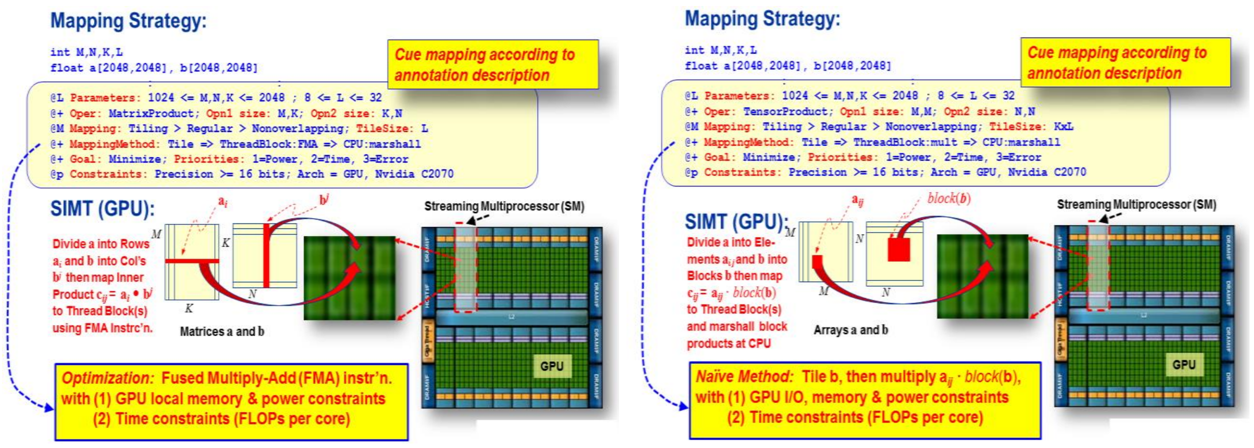
Figure 7. Example of PCEM code generation for convolution: (a) CUDA code and (b) OpenMP code.

3.5. Matrix Product and Tensor (Outer) Product Operations

The operations shown thus far become components for implementation of the more involved matrix product and tensor product operations. For example, the matrix product can be implemented in terms of the inner product. That is, if *A* and *B* are real-valued matrices of respective size *M*x*K* and *K*x*N*, i.e., $A \in R_{MxK}$ and $B \in R_{KxN}$, the matrix product $C = AB$ is defined as

$$(AB)_{ij} = \sum_{k=1}^{K} A_{ik} \cdot B_{kj} \qquad (16)$$

Given row vector A_i and column vector B_j, we have that $(AB)_{ij} = A_i \bullet B_j$, as portrayed in Figure 8a.



```
int M,N,K,L
float a[2048,2048], b[2048,2048]

@L Parameters: 1024 <= M,N,K <= 2048 ; 8 <= L <= 32
@+ Oper: MatrixProduct; Opn1 size: M,K; Opn2 size: K,N
@M Mapping: Tiling > Regular > Nonoverlapping; TileSize: L
@+ MappingMethod: Tile => ThreadBlock:FMA => CPU:marshall
@+ Goal: Minimize; Priorities: 1=Power, 2=Time, 3=Error
@p Constraints: Precision >= 16 bits; Arch = GPU, Nvidia C2070
```

```
int M,N,K,L
float a[2048,2048], b[2048,2048]

@L Parameters: 1024 <= M,N,K <= 2048 ; 8 <= L <= 32
@+ Oper: TensorProduct; Opn1 size: M,M; Opn2 size: N,N
@M Mapping: Tiling > Regular > Nonoverlapping; TileSize: KxL
@+ MappingMethod: Tile => ThreadBlock:mult => CPU:marshall
@+ Goal: Minimize; Priorities: 1=Power, 2=Time, 3=Error
@p Constraints: Precision >= 16 bits; Arch = GPU, Nvidia C2070
```

Mapping Strategy:
Cue mapping according to annotation description
SIMT (GPU):
a_i
b^j
M
K
K
N
Divide a into Rows a_i and b into Col's b^j then map Inner Product $c_{ij} = a_i \bullet b^j$ to Thread Block(s) using FMA Instrc'n.
Matrices a and b
Streaming Multiprocessor (SM)
GPU
Optimization: Fused Multiply-Add (FMA) instr'n. with (1) GPU local memory & power constraints (2) Time constraints (FLOPs per core)
Mapping Strategy:
Cue mapping according to annotation description
SIMT (GPU):
a_{ij}
block(b)
M
N
M
N
Divide a into Elements a_{ij} and b into Blocks b then map $c_{ij} = a_{ij} \cdot block(b)$ to Thread Block(s) and marshall block products at CPU
Arrays a and b
Streaming Multiprocessor (SM)
GPU
Naïve Method: Tile b, then multiply $a_{ij} \cdot block(b)$, with (1) GPU I/O, memory & power constraints (2) Time constraints (FLOPs per core)

(a) (b)

Figure 8. Example of PCEM implementation of (a) matrix product and (b) tensor product on a GPU.

The tensor (outer) product, portrayed in Figure 8b, is similarly implemented, by observing that, given matrices $A \in \mathbb{R}_{MxN}$ and $B \in \mathbb{R}_{KxL}$, the *outer product of A and B*, denoted by $C = A \otimes B$, is defined in image algebra as:

$$(A \otimes B)_{\mathbf{ij}} = a_{ij} * B,\ 1 \leq i \leq M,\ 1 \leq j \leq N. \tag{17}$$

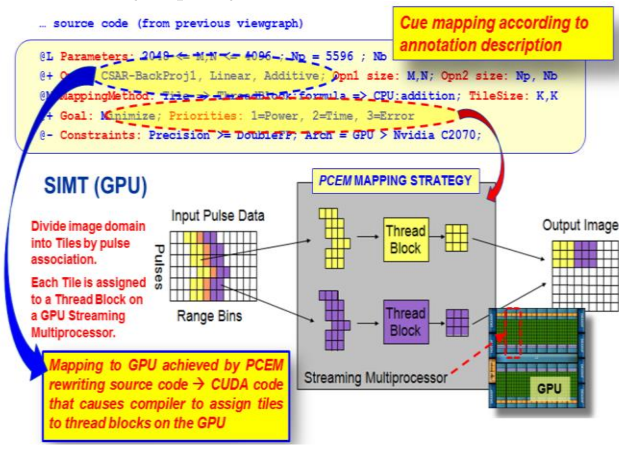
Scheduling of the outer product depends upon block sizes K_A and K_B of the input matrices and their relationship to the blocksize $K_A \cdot K_B$ of the output matrix, as well as the size of local memory in the slave processor, the bus or network communication protocol and cost, and computational cost incurred by the extensive number of multiplies.

4. IMPLEMENTATIONAL EXAMPLE OF PCEM SPECIALIZATION

Advantageously, the PCEM types of parallel operations form a compact set of *operation classes* that can be specialized into *subclasses* and *instances*, as indicated by principles of object-oriented design. Each of these instances carries with it at least one algorithm for mapping its operation to each target architecture, together with load balancing heuristics, constraints for mapping to heterogeneous architectures, and so forth. As such, we have a compact hierarchy of operation classes, subclasses, and instances – each of which can be mapped to a target architecture. This differs significantly from previous work [Dur12], in which a user can code an *ad hoc* procedure or kernel, link it to annotation that can be inserted into application code, then have the annotation processed according to user-specified constraints to yield parallel code. But Duran et al.'s kernels do not seem to be organized systematically, which could lead to undesirable software engineering practice.

PCEM's advantage in this regard is significant: *users do not have to do much coding, and the PCEM system takes care of partitioning and scheduling automatically*. PCEM can do this efficiently and successfully because the execution, mapping, and scheduling methods (or knowledge) are encapsulated with each operation instance that comprises the hierarchy of the PCEM parallel virtual machine. The key concept is *specialization* that can inherit attributes and methodology from parent (*class* level) operation annotation and procedures.

From Section 3.5, we note that the tensor product (or outer product) involves the multiplication of two matrices A and B in blockwise fashion. Equation (17) shows that the $(i,j)^{\text{th}}$ block of the tensor product $C = A \otimes B$ is formed by pointwise multiplication of the element a_{ij} of matrix A by the matrix B. In order to optimize performance of this operation we must partition A and B blockwise (e.g., by tiling) and optimize the blocksizes with respect to processor local memory capacity and data movement latencies.



```
… source code (from previous viewgraph)
@L Parameters: 2048 <= M,N <= 4096 ; Np = 5596 ; Nb
@+ Opn: CSAR-BackProj1, Linear, Additive; Opn1 size: M,N; Opn2 size: Np, Nb
@M MappingMethod: Tile => ThreadBlock:formula => CPU:addition; TileSize: K,K
@+ Goal: Minimize; Priorities: 1=Power, 2=Time, 3=Error
@- Constraints: Precision >= DoubleFP; Arch = GPU > Nvidia C2070;
```

Cue mapping according to annotation description
SIMT (GPU)
PCEM MAPPING STRATEGY
Divide image domain into Tiles by pulse association.
Each Tile is assigned to a Thread Block on a GPU Streaming Multiprocessor.
Input Pulse Data
Pulses
Range Bins
Thread Block
Thread Block
Output Image
Streaming Multiprocessor
GPU
Mapping to GPU achieved by PCEM rewriting source code → CUDA code that causes compiler to assign tiles to thread blocks on the GPU

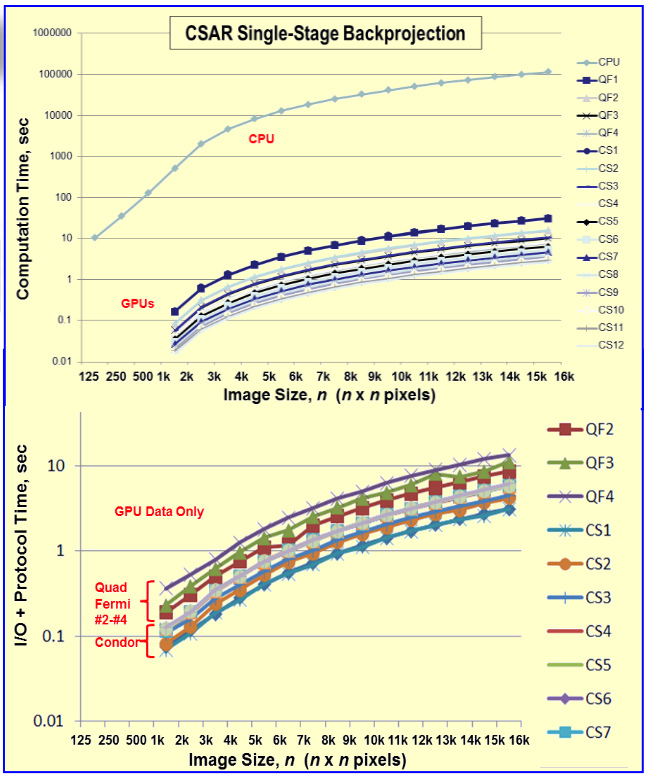


Figure 9. CSAR image reconstruction: (a) Mapping technique derived from the tensor product (Fig. 8b); (b) Measured performance results, from AFRL network-connected Condor machine (CS*x* notation) and our bus-connected Nvidia quad Fermi machine (QF*x*) notation. Observe how the I/O protocol time contributes a significant amount of the total computation time in the quad Fermi machine (enhanced PCI bus), as opposed to reduced I/O time in Condor, due to faster network connection (Infiniband$^{\text{TM}}$).

Single stage CSAR image reconstruction poses a similar problem: each pulse-azimuth value in a radar pulse data matrix influences the reconstruction of each pixel in a two-dimensional output image. The compact PCEM annotation for this process is presented below. The logical sub-block (@L) describes the image size constraints (M,N) and the pulse matrix size limits (N_b and N_p) illustrated in Figure 9a, together with the CSAR backprojection operation as *linear* and *additive*. Due to the process of specialization (of the tensor product operation type), the annotation is of similar form to that of the tensor product (Figure 8b):

```
@D PCEM: Example of Backprojection
@L Oper: Type=Backprojection;
@+ Precision=32;
@+ Opn1: Datatype=cfloat; Domain=rectangular; Size1=bins; Size2=pc; Ptr=p0; df=df
@+ Opn2: Datatype=float; Domain=rectangular; Size1=5; Size2=pc; Ptr=m0;
@+ Dest: Datatype=cfloat; Domain=rectangular; Size1=s0; Size2=s1; Ptr=out;
       x0=-30; x1=30; y0=-30; y1=30
```

PCEM's algorithm exercising capability was applied to the preceding annoration, to produce performance data that are summarized in Figure 9b. There, an expected, significant discrepancy exists between the NVIDIA Tesla C2050 GPU and Intel dual core CPU execution times. Further, using PCEM's exercising scripts, we measured the effect on the GPUs of only I/O and protocol delays. We found that, as the number of processors active on the Condor system increases, and image size increases, the (I/O + protocols) cost overwhelms the computation cost.

For example, for two Condor GPUs active, the total runtime is 1.189 sec for a 4kx4k-pixel output image and 16.32 sec for a 15kx15k image, whereas for 12 Condor GPUs active, the total runtime is 0.204 sec for a 4kx4k output image and 2.73 sec for a 15kx15k image. In contrast, given the (I/O + protocols) times, for 2 Condor GPUs active, the time is 0.35 sec for a 4kx4k output image and 4.23 sec for a 15kx15k image, whereas for 12 Condor GPUs active, the time is 0.507sec for a 4kx4k output image and 6.234 sec for a 15kx15k image. Thus, the ratio of runtime to I/O+protocol time increases from 3.37X = 1.189s/0.35s with 2 processors and a 4kx4k image, to 3.86X = 16.32s/4.23s for 2 processors reconstructing a 15kx15k image. A more dramatic *decrease* in runtime divided by I/O+protocol time, *with respect to the two-GPU Condor case* is seen for 12 processors, for 4kx4k image we have 0.402X = 0.204s/0.507s but for the 15kx15k image, we have 0.44X = 2.73s/6.23s. We are further investigating these effects of network overhead, which we are attempting to remediate through more carefully optimized data movement strategies.

6. CONCLUSIONS AND FUTURE WORK

We have presented an annotation and optimizing scheduler driven by annotation (higher-level algorithmic expression) that is applicable to envisioned exascale architectures. Specifically, PCEM specifies and directs assignment of computational tasks to multiple target processors based on annotation-directed tradeoffs between parallelism and locality. The optimizing scheduler is designed to be directed by constraints specified in the annotation. We provided a CSAR image reconstruction example to show that PCEM achieves good runtime performance by increasing parallelism adaptively through (a) partitioning and coalescing of data (operands) and tasks (operations) to reduce I/O cost by intelligent management of locality, (b) automatic tiling of the pulse matrix and reconstructed image to exploit available parallelism in the target GPUs, and (c) superior performance with respect to high-level language versions of the program run on multicore CPUs.

The PCEM concept is built on the mathematical models underlying University of Florida's development of image algebra, a rigorous, concise notation that unifies linear and nonlinear mathematics in the image domain. The PCEM optimizing scheduler is built on customary optimization theory. Additionally, we have developed theoretical expressions that describe how computational resources will be allocated to achieve desired tradeoffs between *parallelism* (which we can increase to achieve decreased runtime) and *locality* (we can more tightly cluster computations in fewer and more proximal processors, to achieve decreased power consumption).

Hybrid multicore processors (HMPs) and graphics processing units (GPUs) are leading a new wave of computing that combines massive parallelism with MIMD parallel networks of CPUs. As programmers learn to develop inherently parallel algorithms using the PCEM paradigm, and users share the joy of using PCEM (usually without knowing why things are faster and better), the deployment of hybrid MIMD/SIMD architectures as stand-alone or closely coupled with a resident CPU will explode. HMP and GPU demand will grow much faster than the PC market or the much-vaunted mobile market, thereby becoming the next big wave in computing. Thus, PCEM positions us to provide intelligent scheduling services as this trend reaches the exascale regime.

7. REFERENCES

[Amd67] Amdahl, Gene (1967). "Validity of the single processor approach to achieving large-scale computing capabilities" *AFIPS Conference Proceedings* **30**: 483–485.

[Cass12] Cassidy, A.S. and A.G. Andreou (2012) "Beyond Amdahl's Law: An object function that links multiprocessor performance gains to delay and energy", *IEEE Trans. Computers*, **61**:1110-1126.

[Chap11] Chapman, W., S. Ranka, S. Sahni, and M. Schmalz. (2011) "Parallel processing techniques for the processing of synthetic aperture radar data on GPUs", in *Proceedings of the IPDPS 2011 Conference*.

[Dur12] Duran, A., E. Ayguadé, R.M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. (2012) "OmpSs: A proposal for programming heterogeneous multi-core architectures", *Par. Proc. Lett.* **21**:173.

[Gil11] Gillespie, C.S. (2011). "CPU and GPU trends over time", Web, http://csgillespie.wordpress.com/ /2011/01/25/cpu-and-gpu-trends-over-time/ **.**

[Jack97] Jackson, R.D., P.C. Coffield, and J.N. Wilson. (1997) "A new SIMD computer vision architecture with image algebra programming environment", *Proc. IEEE 1997 Aerospace Conference*, pp. 169-185.

[Int12] Intel, Inc. (2012) "Intel 22nm Technology", Web: http://www.intel.com/content/ www/us/en/silicon-innovations/intel-22nm-technology.html.

[Li09] Li, Y., S. Ranka, S. Sahni, and M. Schmalz. (2009) "Network centered multiple resource scheduling in e-science applications", *GridNets*, 2009, LNICST **25**: 37-44, Springer Verlag.

[Li12] Li, J., S. Ranka and S. Sahni. (2012) "GPU Matrix Multiplication," *Multi- and Many-Core Technologies: Architectures, Programming, Algorithms, and Applications*, Chapman Hall/CRC, Ed. S. Rajasekaran.

[Lul11] Luley, R., C. Usmail, and M. Barnell. (2011) "Energy efficiency evaluation and benchmarking of AFRL'S Condor high performance computer", *Proceedings of the DoD High Performance Modernization Program's 2011 User Group Conference*, Portland OR.

[Ped12] Peddie, J. (2012) "Graphics shipments in Q2 increased", Press release, Jon Peddie MarketWatch, Website, http://jonpeddie.com/press-releases/details/graphics-shipments-in-q2-increased-2.5-over-last-quarter-and-5.5-over-last-/ .

[Rit01] Ritter, G.X. and J.N. Wilson. (2001) *Handbook of Computer Vision Algorithms in Image Algebra*, Second Edition, Boca Raton, FL: CRC Press.

[Sch10] Schmalz M.S., G.X. Ritter, and E. Hayden. (2010) "Image algebra Matlab 2.3 and its application to research in image processing and compression", in *Proc. SPIE* **7799**.

[Sch11] Schmalz, M.S., G.X. Ritter, E. Hayden, and G. Key. "Algorithms for adaptive nonlinear pattern recognition", in *Proceedings SPIE* **8136** (2011).

[Swe98] Sweat, M. and J.N. Wilson. (1998) "Overview of AIM: Supporting computer vision on heterogeneous high-performance computing systems", *Proceedings SPIE* **3452**:81.

[Wil88] Wilson, J., G. Fischer, and G. Ritter. (1988) "Implementation and use of an image processing algebra for programming massively parallel computers,", in *Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation*, (Fairfax VA), pp. 587-594.

[Wil89] Wilson, J., D. Wilson, G. Ritter, and D. Langhorne. (1989) "Image algebra FORTRAN language, version 3.0," Tech. Rep. TR-89-03, University of Florida CIS Department, Gainesville, FL.

[Wil91] Wilson, J. (1991) "An introduction to image algebra Ada," in *Image Algebra and Morphological Image Processing II, Proceedings SPIE* **1588**:101-112.

[Wil93] Wilson, J. (1993) "Supporting image algebra in the C++ language," in *Image Algebra and Morphological Image Processing IV, Proceedings SPIE* **2030**:315-326.

[Wil97] Wilson, J. and E. Riedy. (1997) "Efficient SIMD evaluation of image processing programs," in *Parallel and Distributed Methods for Image Processing, Proceedings SPIE* **3166**:199-211.