A Principled Kernel Testbed for Hardware/Software Co-Design Research

Alex Kaiser, Samuel Williams, Kamesh Madduri, Khaled Ibrahim, David Bailey, James Demmel, Erich Strohmaier
{ ADKaiser, SWWilliams, KMadduri, KZIbrahim, DHBailey, EStrohmaier }@lbl.gov, demmel@eecs.berkeley.edu

FUTURE TECHNOLOGIES GROUP

Motivation

Current research is focused on how to effectively use an ever diversifying array of parallel processors. As such, the community is being driven into an evolutionary and architecturally-driven mindset. We believe this will yield suboptimal results.

For hardware/software co-design to truly be effective, we must start from the core computational methods we wish to accelerate, not code extracted from existing applications.

Thus, this project is focused on creating a kernel testbed based on the core computational methods found in high-performance computing. We believe the core methodology (if not some of the kernels) are applicable in other domains. Previous attempts have create benchmarks that may not fully enable inter-disciplinary research.

Benchmark limitations vs. fields of research enabled by the benchmark

Benchmark Style	Micro-arch.	Compilers	Instruction Set	SW Optimization	Prog. Models	Languages	Memory Arch.	Algorithms
	Enabled Fields of Research							
Fixed Binary	✔							
Fixed Source Code	✔	✔	✔					
Fixed Interface, but may optimize code	✔	✔	✔	✔				
Code-Based Problem Definition	✔	✔	✔	✔	✔			
High-Level Problem Definition	✔	✔	✔	✔	✔	✔	✔	✔

Intended Usage

- We use the taxonomy that researchers should produce a HW/SW ***"solution"*** that efficiently implements the ***"problem"*** as specified using a domain-specific language.
- We believe researchers will be able to take our testbed and create benchmarks that foster research in many fields.
- One may gauge the quality of the solution through a variety of existing metrics based on performance, energy, power, cost, productivity, etc…

Testbed Components

- Our testbed is composed of a series of kernels.
- For each kernel, the testbed mandates creation of:
  1. a formal problem **specification** in a mathematical, or domain-appropriate language
  2. a scalable **input generator**
  3. a scalable **verification** scheme
- Optionally, we provide a reference implementation in commonly used programming languages.
- Additionally, we may provide an optimized reference implementation that provides insights into the bottlenecks on existing hardware and researcher's optimizations to eliminate, hide, or mitigate them.

(1) Problem Specification

- The problem specification for a kernel mathematically or quantitatively defines the functional relationship between input and output.
- We strive not to use array notation or other programming language-based constructs (e.g. loops for parallel constructs) in our definitions.
- For example, in numerical linear algebra, we define problems using the well developed lexicon of operands (scalars, vectors, matrices) and operators (addition, multiplication, transpose, inverse, summation, etc…)

(2) Scalable Input Dataset

- Wherever possible, each kernel problem definition is accompanied by a scalable input generation scheme
- They should be amenable to straightforward and independent verification while guaranteeing the existence of a solution (random inputs may not suffice).
- When performing distributed or novel HW/SW design, researchers should re-implement the input generators.

(3) Verification Scheme

- We wish to verify problems independently from their definitions. One shouldn't use reference codes/HW to verify novel hardware/software designs.
- In many domains, for carefully constructed inputs, we may provide an analytic solution based on the calculus of the underlying mathematics.
  (see example in next column)
- Some kernels are simple functions (they're not solvers). For them, complex verification schemes are usually not needed.

Reference Implementation

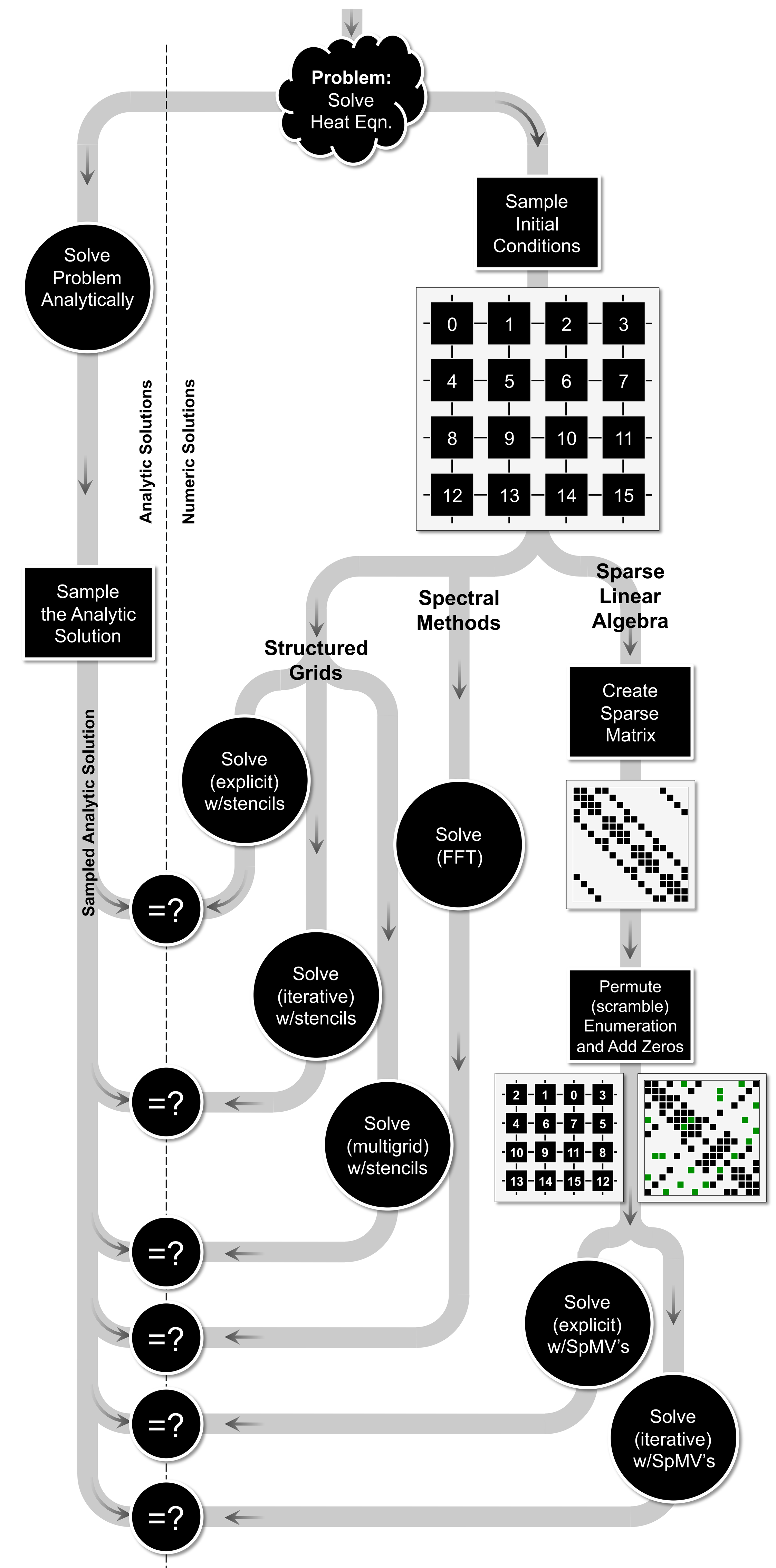
- To provide some guidance as how one might implement such a kernel using existing languages, programming models, and hardware, we provide a reference implementation for each kernel.
- The reference implementation is either a sequential C or MATLAB program including the input generation and verification components (where applicable)
- **The reference implementations should never be used as the basis for benchmarking. It is incumbent upon researchers to produce appropriate implementations for their field of research.**

Quality of HW/SW Solution

- If this testbed were used only for SW optimization, then the quality of the optimized implementations is primarily time or energy.
- If used for HW/SW co-design, hardware design cost and portability should be considered
- If used for programming model or language research, productivity might be of interest.

Input / Verification Example

- Consider solving the heat equation PDE on a rectangular N-dimensional domain.
- By carefully selecting the initial and boundary conditions, we may analytically solve the problem.
- Conversely, we may solve the problem numerically using one of six different methods (spanning three dwarfs)
- All methods should produce the same answer as a sampling of the analytic solution.
- We may aggressively push the complexity in the sparse arena by permuting the grid enumeration (rows/ columns) or randomly adding explicit zeros.



Our Kernel Testbed Today

- To date, we have created a testbed of over 40 kernels
- Virtually every non-trivial kernel has an associated scalable verification scheme.
- Additionally, we have created sequential C or MATLAB reference implementations for most of them.
- We list their status below and categorize them based on the original seven dwarfs or Berkeley's subsequent 13 Motifs.

■ *Dwarf of primary kernel*
□ *Dwarf of supporting kernels*

Kernel	Dense Linear Alg.	Sparse Linear Alg.	Structured Grids	Unstructured Grids	Spectral	Particles	Monte Carlo	Graphs and Trees	Sorting	Kernel Definition	Reference Implementation	Optimized Implementation	Scalable Inputs	Verification Scheme
Scalar-Vector Multiplication	■									✔	✔		✔	
Elementwise-Vector Mult.	■									✔	✔		✔	
Matrix-Vector Mult.	■									✔	✔		✔	
Matrix-Matrix Mult.	■									✔	✔		✔	
LU Factorization	■									✔	✔	✔	✔	✔
Symmetric Eigensolver (QR)	■									✔	✔		✔	✔
Cholesky Factorization	■									✔	✔		✔	
SpMV (y=Ax)		■								✔	✔	✔	✔	✔
SpTS (Lx=b)		■								✔	✔		✔	✔
Matrix Powers ($y_k=A^kx$)		■								✔	✔		✔	✔
Conjugate Gradient	□	■								✔	✔		✔	✔
KSM/GMRES	□	■								✔	✔		✔	✔
SpLU	□	■												
Finite Difference Derivatives			■							✔	✔		✔	✔
FD/Laplacian			■							✔	✔	✔	✔	✔
FD/Gradient			■							✔	✔	✔	✔	✔
FD/Divergence			■							✔	✔	✔	✔	✔
FD/Curl			■							✔	✔		✔	✔
FD/Solve PDE (explicit)			■							✔	✔		✔	✔
FD/Solve PDE (implicit)	□		■							✔	✔		✔	✔
FD/Solve PDE (multigrid)	□		■							✔	✔		✔	✔
There are a number of other important structured grid methods including lattice Boltzmann (LBM), finite volume, and AMR that we have yet to enumerate representative kernels for.														
Although even within our community unstructured grids are commonly used, we have yet to enumerate any concise representative kernels.														
1D FFT (complex → complex)					■					✔	✔		✔	✔
3D FFT (complex → complex)					■					✔	✔		✔	✔
Convolution	□				■					✔	✔		✔	✔
Solve PDE via FFT	□		■		■					✔	✔		✔	✔
2D N^2 Direct						■				✔	✔		✔	
3D N^2 Direct						■				✔	✔		✔	
2D N^2 Direct (with cut-off)						■				✔	✔		✔	✔
3D N^2 Direct (with cut-off)						■				✔	✔		✔	✔
2D Particle-in-cell (PIC)			□			■								
3D Particle-in-cell (PIC)			□			■								
2D Barnes Hut						■		□		✔	✔		✔	✔
3D Barnes Hut						■		□		✔	✔		✔	✔
2D Fast Multipole Method					□	■		□						
3D Fast Multipole Method					□	■		□				✔		
Quasi-Monte Carlo Integration							■			✔	✔		✔	✔
EP Summation							■			✔	✔		✔	✔
Graph Traversal								■		✔	✔		✔	✔
Betweenness Centrality								■		✔	✔		✔	✔
Integer Sort									■	✔	✔		✔	✔
100 Byte Sort									■	✔	✔		✔	✔
Spatial Sort									■	✔	✔		✔	✔

Our kernel selection predominantly reflects scientific computing applications. There are numerous other application domains within computing whose researchers should enumerate their own representative problems. Some of the problems from other domains may be categorized using the aforementioned motifs, some may be categorized into other Berkeley Motifs not listed above (such as branch-and-bound, or dynamic programming), while others may necessitate novel motif creation.