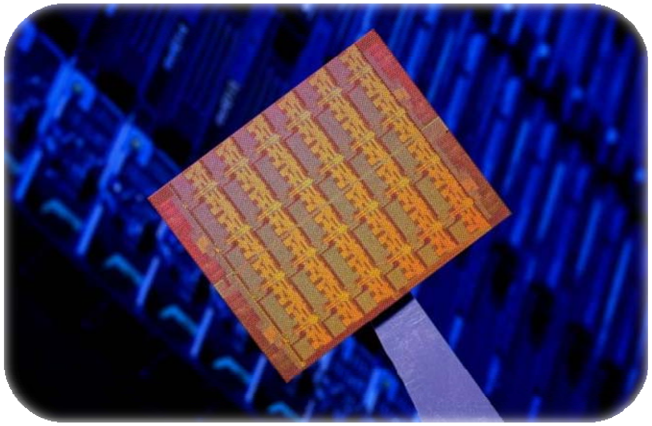
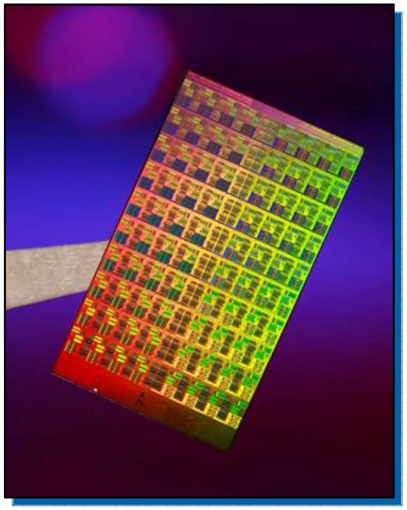


The Future of Many Core Computing:
Software for many core processors



Tim Mattson
Intel Labs
January 2010

Disclosure

- The views expressed in this talk are those of the speaker and not his employer.
- I am in a research group and know nothing about Intel products. So anything I say about them is highly suspect.
- This was a team effort, but if I say anything really stupid, it’s all my fault … don’t blame my collaborators.

Sources and Acknowledgments

- Slides from my work at Intel.

- Slides I created with “UC Berkeley ParLab colleagues”. Most of these come from courses I taught with Prof. Kurt Keutzer.

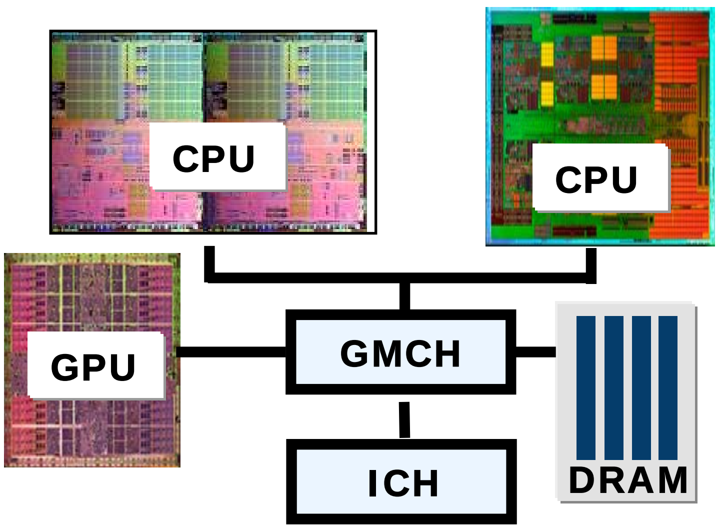
- Slides I developed with members of the Khronos compute group (OpenCL).

Agenda

- The many core software challenge
- OpenCL: a brief overview
- Going beyond OpenCL

Heterogeneous computing

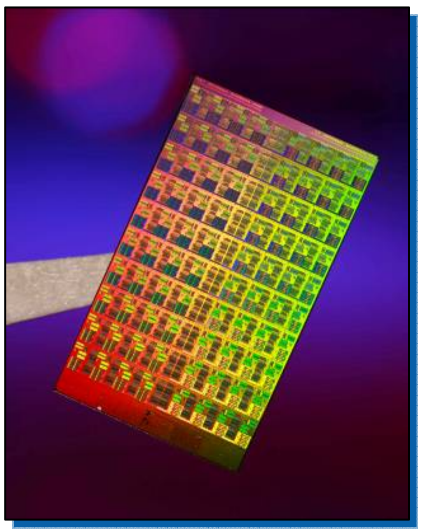
- A modern platform has:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - ... other?



- Programmers need to make the best use of <u>all</u> the available resources from within a <u>single</u> program:
  - One program that runs well (i.e. reasonably close to “hand-tuned” performance) on a heterogeneous mixture of processors.

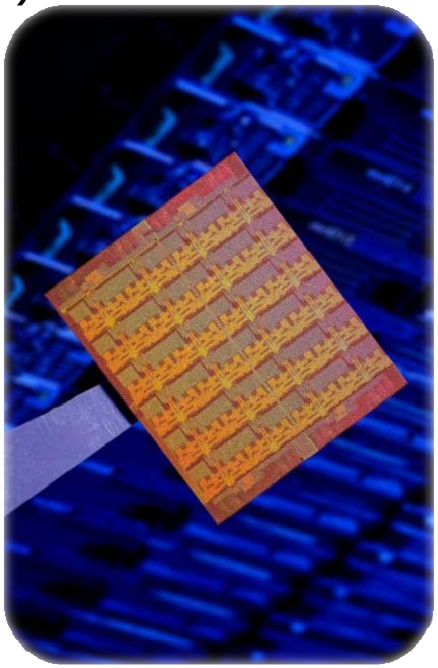
… and many core chips make it worse

- **Scalable architectures research**:
  - How should we connect the cores so we can scale as far as we need (O(100's to 1000) should be enough)?



80 core Research processor

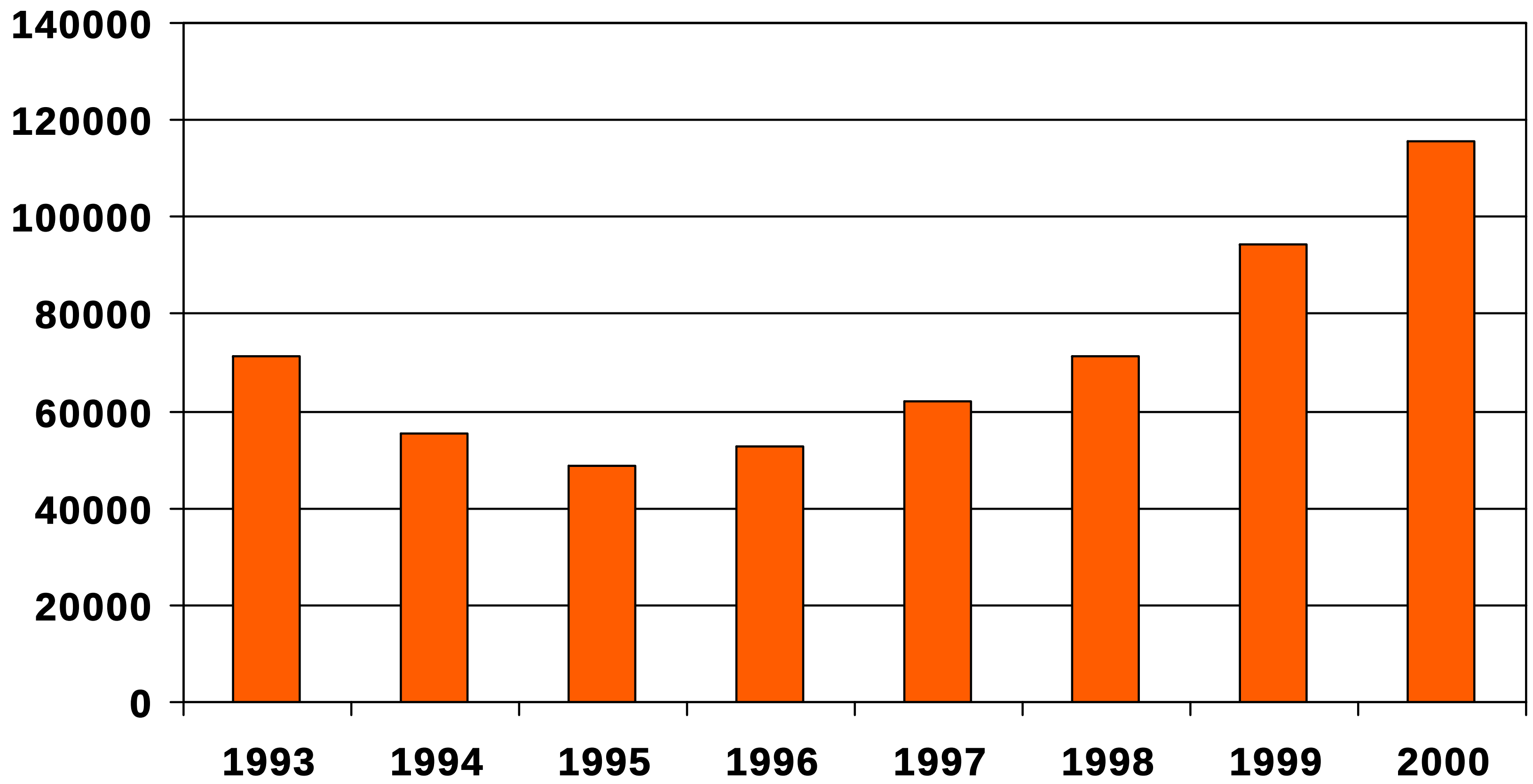
Intel's "TeraScale" processor research program is addressing the question … What is the architecture of future many core chips, and how will we use them.



48 core SCC processor

Parallel hardware trends

Top 500: total number of processors (1993-2000)

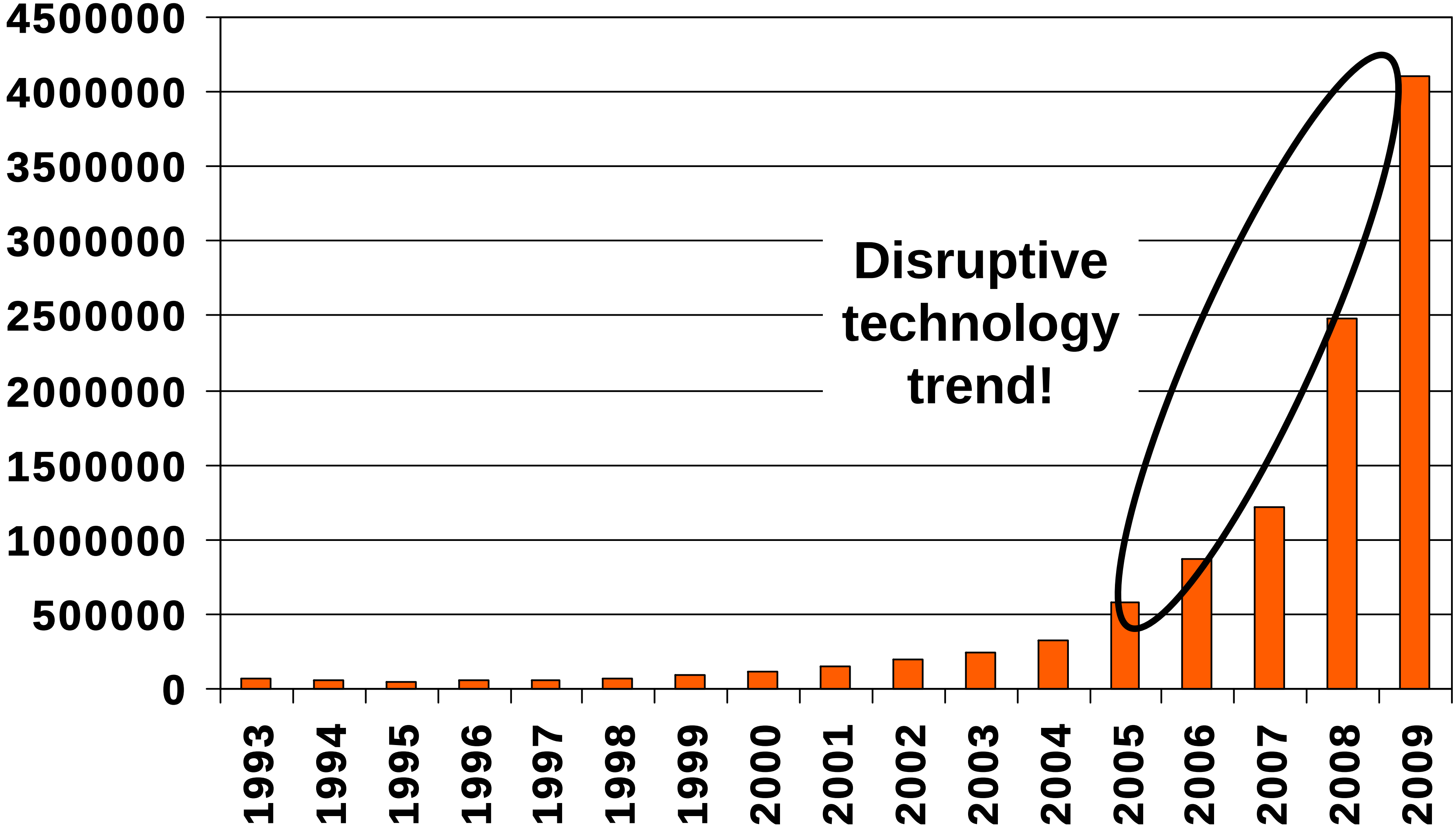


The early years … SIMD MPP fades and clusters take over.

Source: the "June lists" from www.top500.org

Parallel Hardware Trends

Top 500: total number of processors (1993-2009)



Source: the "June lists" from www.top500.org

The many-core challenge

We have arrived at many-core solutions not because of the success of our parallel software but because of our failure to keep increasing CPU frequency.*

- Result: a fundamental and dangerous mismatch
  - Parallel hardware is ubiquitous.
  - Parallel software is rare

Our challenge … make parallel software as routine as our parallel hardware.

*Tim Mattson … a famous-software-researcher wannabe

Solution: Find A Good parallel programming model, right?

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
PCN
PCP:
PH
PEACE
PCU
PET
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM
pC++
SCHEDULE
SciTL
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
XENOOPS
XPC
Zounds
ZPL

Models from the golden age of parallel programming (~95)

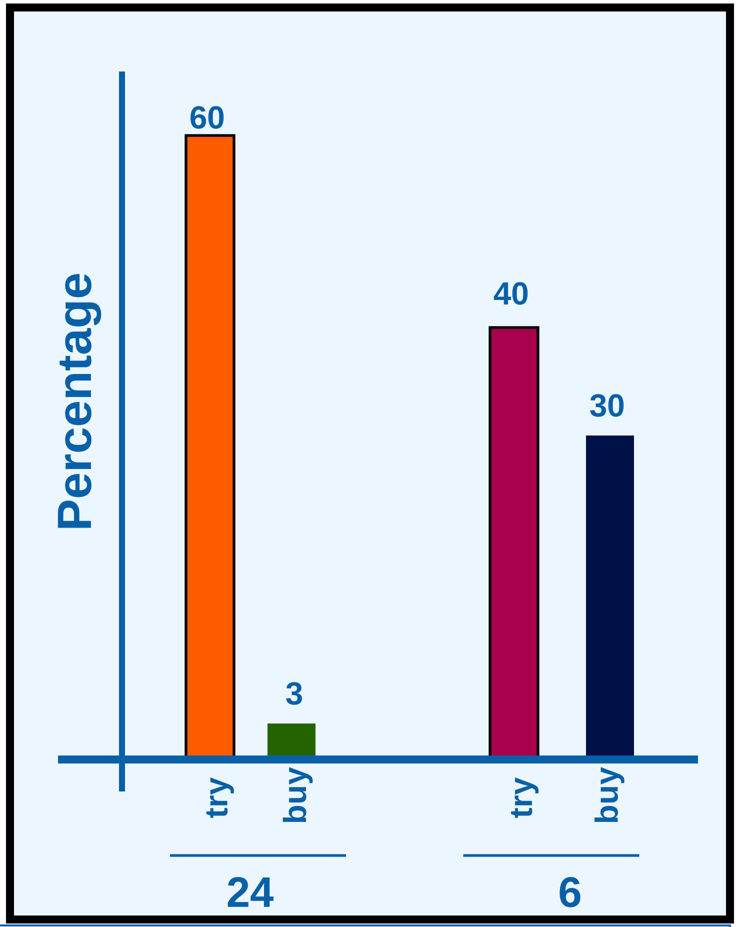
The only thing sillier than creating too many models is using too many

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
PCN
PCP:
PH
PEACE
PCU
PET
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM
pC++
SCHEDULE
SciTL
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
XENOOPS
XPC
Zounds
ZPL

Programming models I've worked with.

Choice overload:
Too many options can hurt you

- The Draeger Grocery Store experiment consumer choice :
  - Two Jam-displays with coupon's for purchase discount.
    - 24 different Jam's
    - 6 different Jam's
  - How many stopped by to try samples at the display?
  - Of those who "tried", how many bought jam?



Programmers don't need a glut of options ... just give us something that works OK on every platform we care about. Give us a decent standard and we'll do the rest

The findings from this study show that an extensive array of options can at first seem highly appealing to consumers, yet can reduce their subsequent motivation to purchase the product.

Iyengar, Sheena S., & Lepper, Mark (2000). When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 76, 995-1006.

How to program the heterogenous platform? Let History can be our guide … consider the origins of OpenMP …

intel

SGI

Cray

Merged, needed commonality across products

KAI

ISV - needed larger market

ASCI

was tired of recoding for SMPs. Forced vendors to standardize.

Wrote a rough draft straw man SMP API

Other vendors invited to join

DEC

HP

IBM

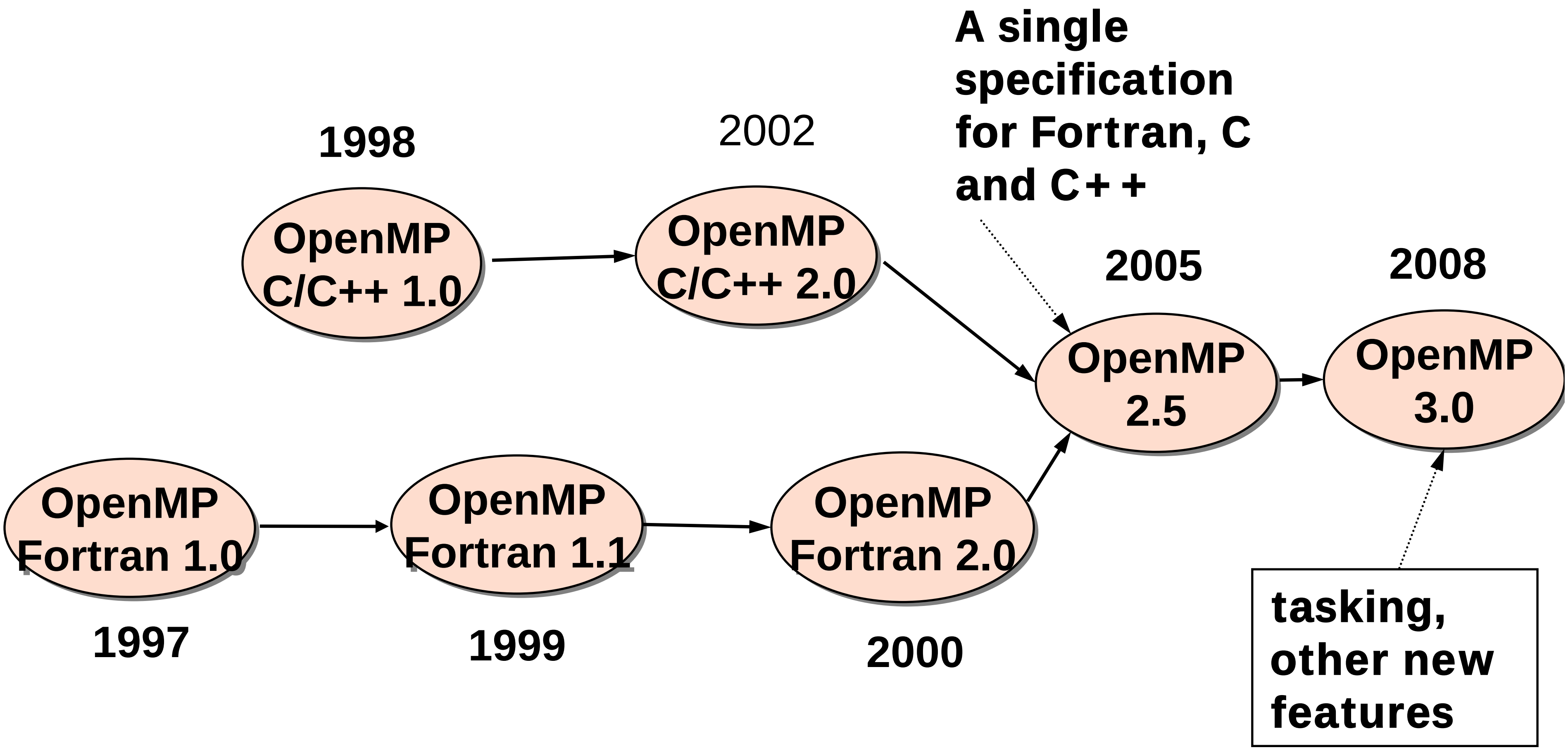
Intel

OpenMP

1997

Third party names are the property of their owners.

OpenMP Release History



OpenCL: Can history repeat itself?

As ASCI did for OpenMP, Apple is doing for GPU/CPU with OpenCL

AMD

ATI

Merged, needed commonality across products

Nvidia

GPU vendor - wants to steel mkt share from CPU

Intel

CPU vendor - wants to steel mkt share from GPU

Apple

was tired of recoding for many core, GPUs. Pushed vendors to standardize.

Wrote a rough draft straw man API

Khronos Compute group formed

Ericsson

Nokia

IBM

Sony

Blizzard

Freescale

TI

+ many more

OpenCL

Dec 2008

Agenda

- The many core software challenge
- OpenCL: a brief overview
- Going beyond OpenCL

OpenCL Working Group

- Diverse industry participation …
  - HW vendors (e.g. Apple), system OEMs, middleware vendors, application developers.
- OpenCL became an important standard “on release” by virtue of the market coverage of the companies behind it.

The BIG idea behind OpenCL

- OpenCL execution model … execute a <u>kernel</u> at each point in a problem domain.
  - –E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

Traditional loops

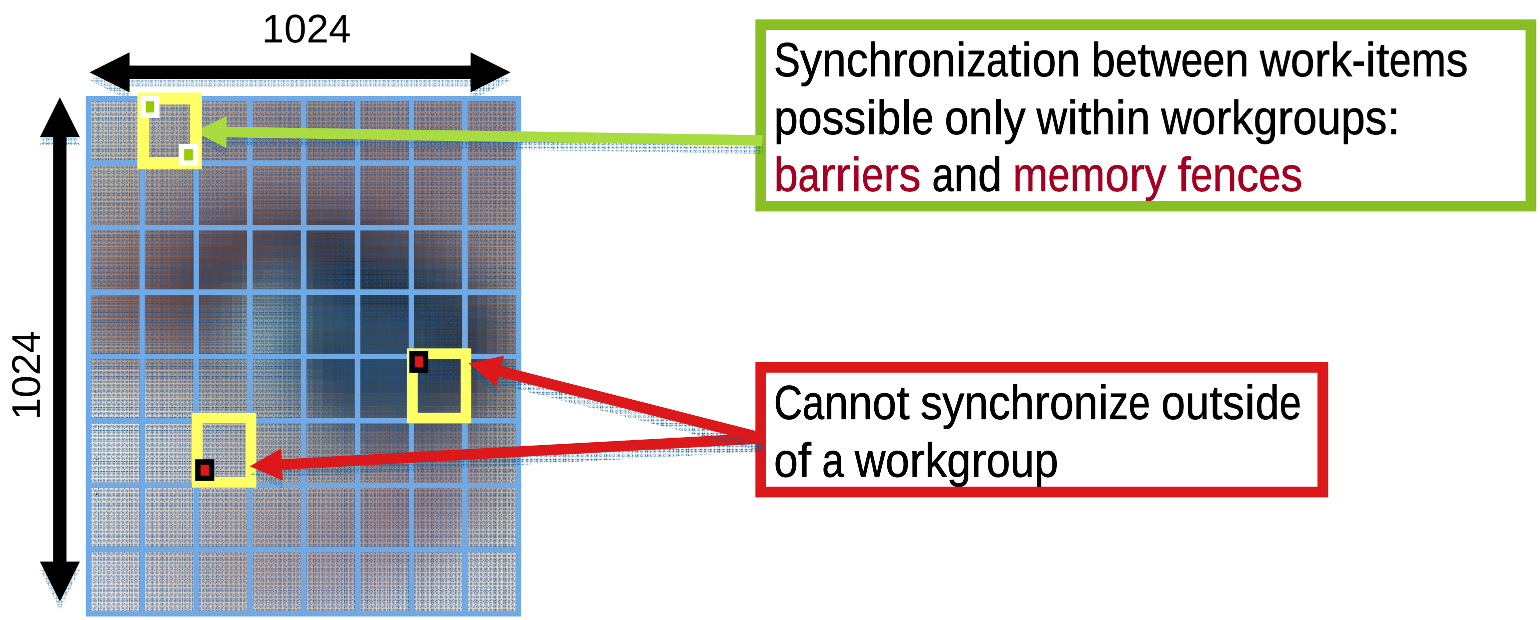
```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

An N-dimension domain of work-items

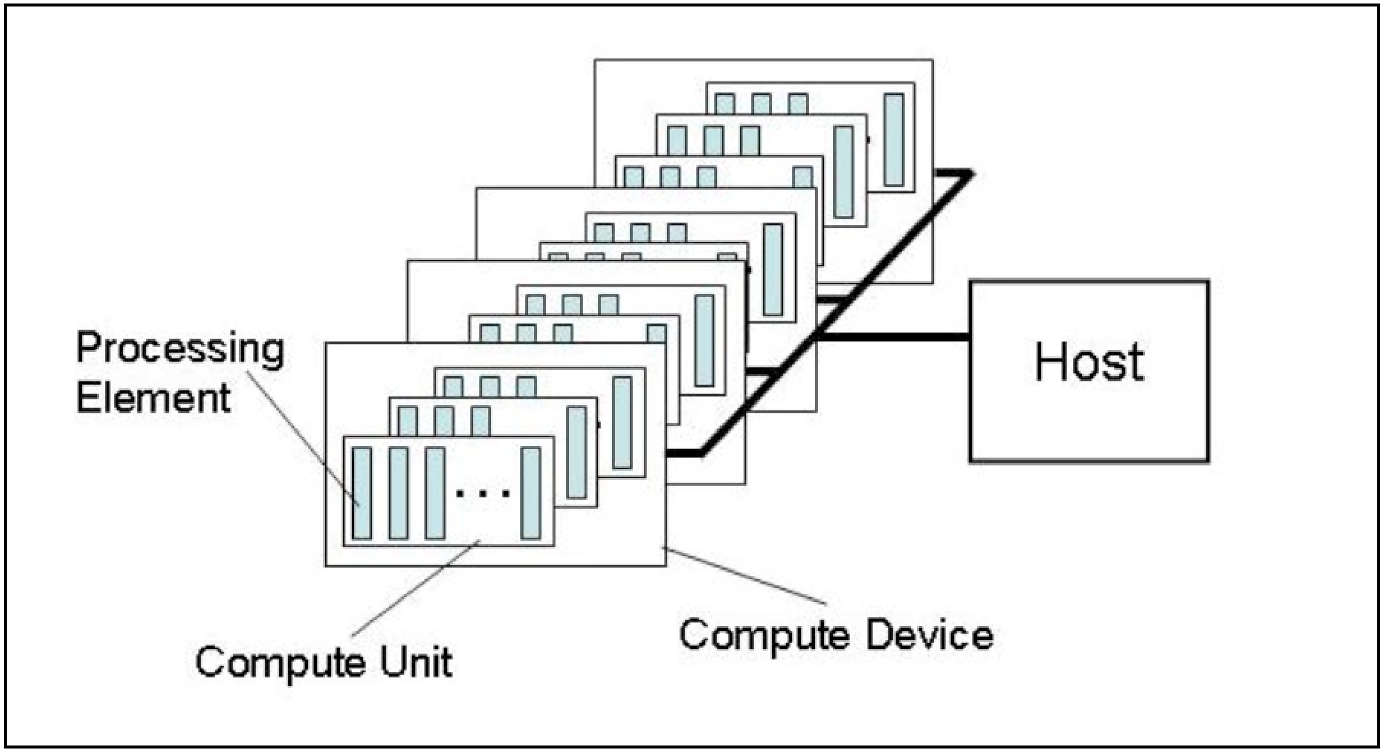
- Define an N-dimensioned index space that is "best" for your algorithm
  - Global Dimensions: 1024 x 1024 (whole problem space)
  - Local Dimensions: 128 x 128 (work group … executes together)



To use OpenCL, you must

- Define the platform
- Execute code on the platform
- Move data around in memory
- Write (and build) programs

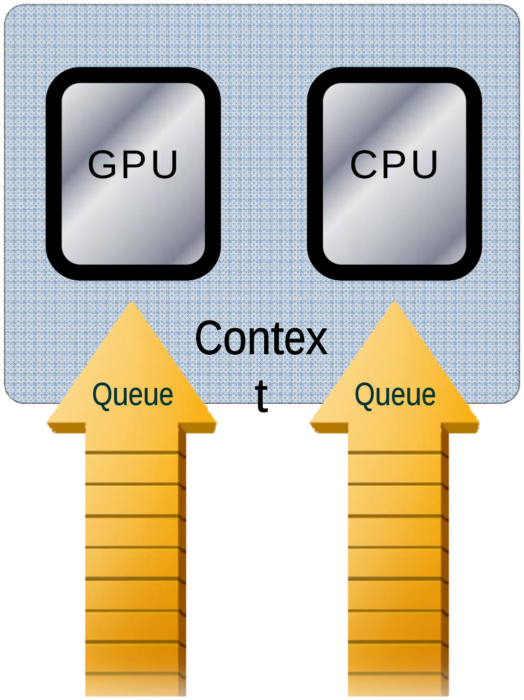
OpenCL Platform Model



- One <u>Host</u> + one or more <u>Compute Devices</u>
  - Each Compute Device is composed of one or more <u>Compute Units</u>
    - Each Compute Unit is further divided into one or more <u>Processing Elements</u>

OpenCL Execution Model

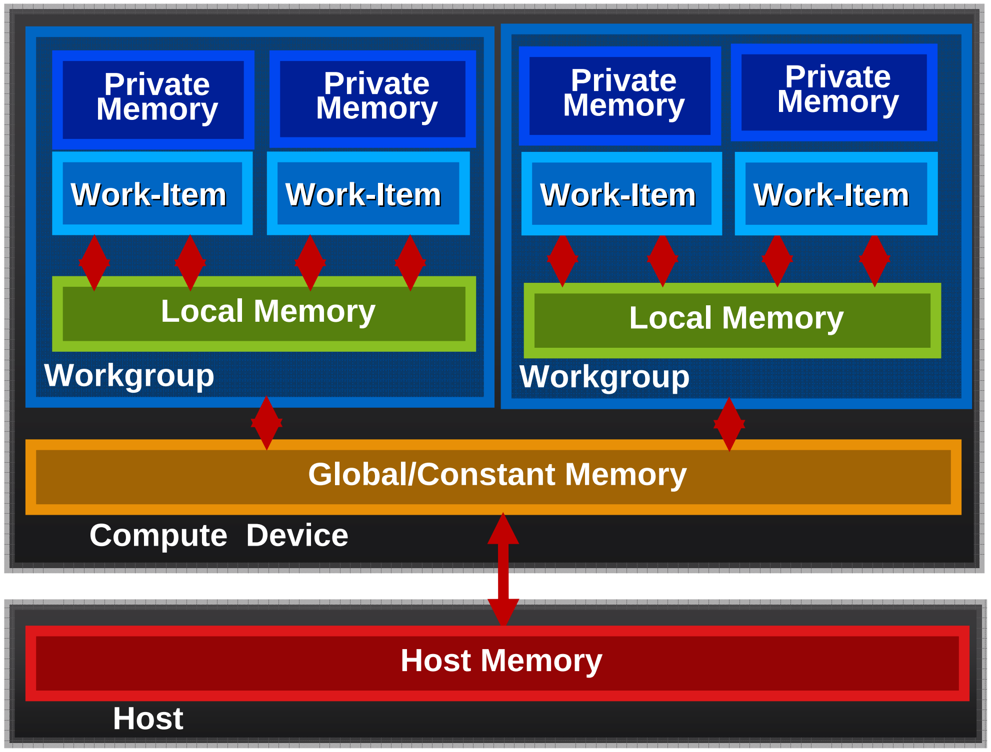
- An OpenCL application runs on a host which submits work to the compute devices.
  - **Work item**: the basic unit of work on an OpenCL device.
  - **Kernel**: the code for a work item. Basically a C function
  - **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
  - **Context**: The environment within which work-items executes … includes devices and their memories and command queues.



- Applications queue kernel execution instances
  - Queued in-order … one queue to a device
  - Executed in-order or out-of-order

OpenCL Memory Model

- **Private Memory**
  - –Per work-item
- **Local Memory**
  - –Shared within a workgroup
- **Global / Constant Memory**
  - –Visible to all workgroups
- **Host Memory**
  - –On the CPU



Memory management is Explicit

You must move data from host -> global -> local ... *and* back

Programming kernels:
the OpenCL C Language

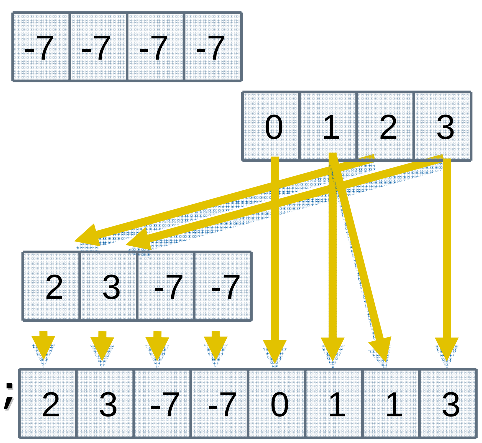
- A subset of ISO C99
  - But without some C99 features such as standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- A superset of ISO C99 with additions for:
  - Work-items and workgroups
  - Vector types
  - Synchronization
  - Address space qualifiers
- Also includes a large set of built-in functions for image manipulation, work-item manipulation, specialized math routines, etc.

Programming Kernels: Data Types

- Scalar data types
  - char , uchar, short, ushort, int, uint, long, ulong, float
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)
- Image types
  - image2d_t, image3d_t, sampler_t
- Vector data types
  - Vector lengths 2, 4, 8, & 16 (char2, ushort4, int8, float16, double2, ...)
  - Endian safe
  - Aligned at vector length
  - Vector operations and built-in functions

Double is an optional type in OpenCL 1.0

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);

vi0.lo = vi1.hi;

int8 v8 = (int8)(vi0, vi1.s01, vi1.odd);
```

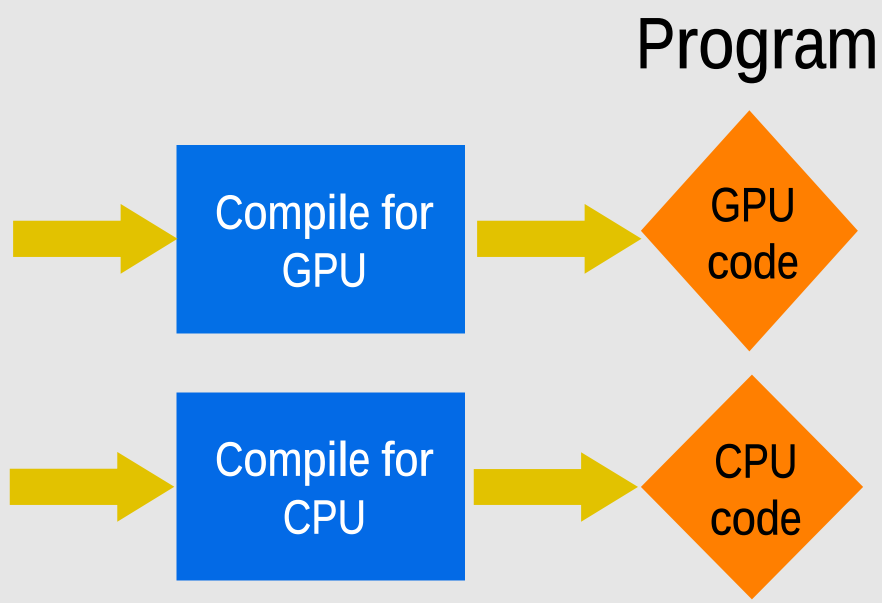


Building Program objects

- The program object encapsulates:
  - A context
  - The program source/binary
  - List of target devices and build options
- The Build process … to create a program object
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

Kernel Code

```
kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                               (int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



Vector Addition - Kernel

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
}
```

Vector Addition: Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with
   context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,
   devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   sizeof(cl_float)*n, srcA,
                                            NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY
   | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
                                            NULL);
memobjs[2] =
   clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                          sizeof(cl_float)*n,
   NULL,
                                            NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                                            NULL);

// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
   CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

Vector Addition: Host Program

OpenCL

Define platform and queues

```
    devices[0], 0, NULL);

// allocate the buffer memory objects
                                        TR,
```

Define Memory objects

```
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY
    | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
    NULL);
memobjs[2] =
    clCreateBuffer(context,CL_MEM_WRITE_ONLY,
```

Create the program

Build the program

Create and setup kernel

```
/
k                                    LL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
// set work-item dimensions
global_w

// execu
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);
```

Execute the kernel

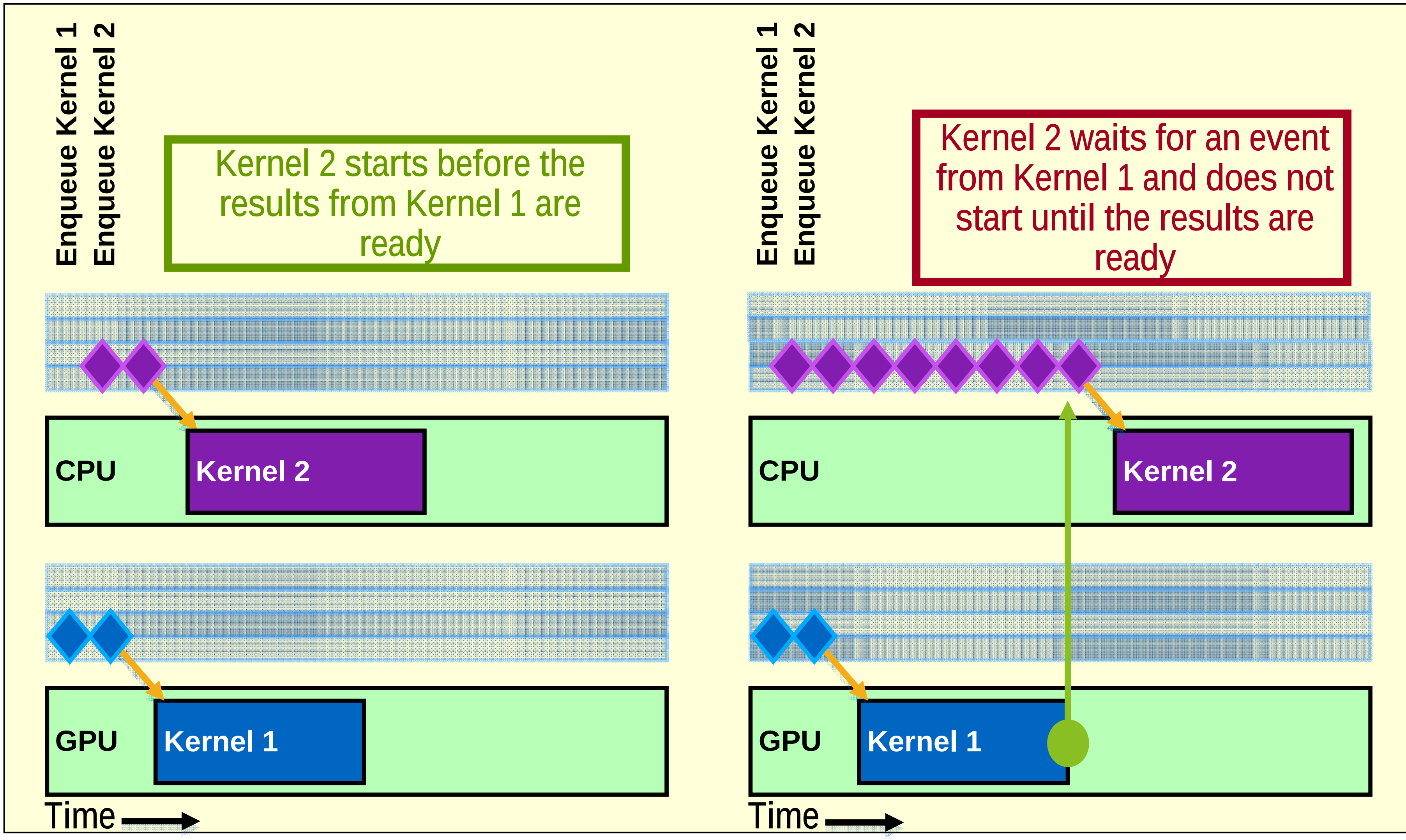
```
// rea
err =                                    TRUE,
   0,
```

Read results on the host

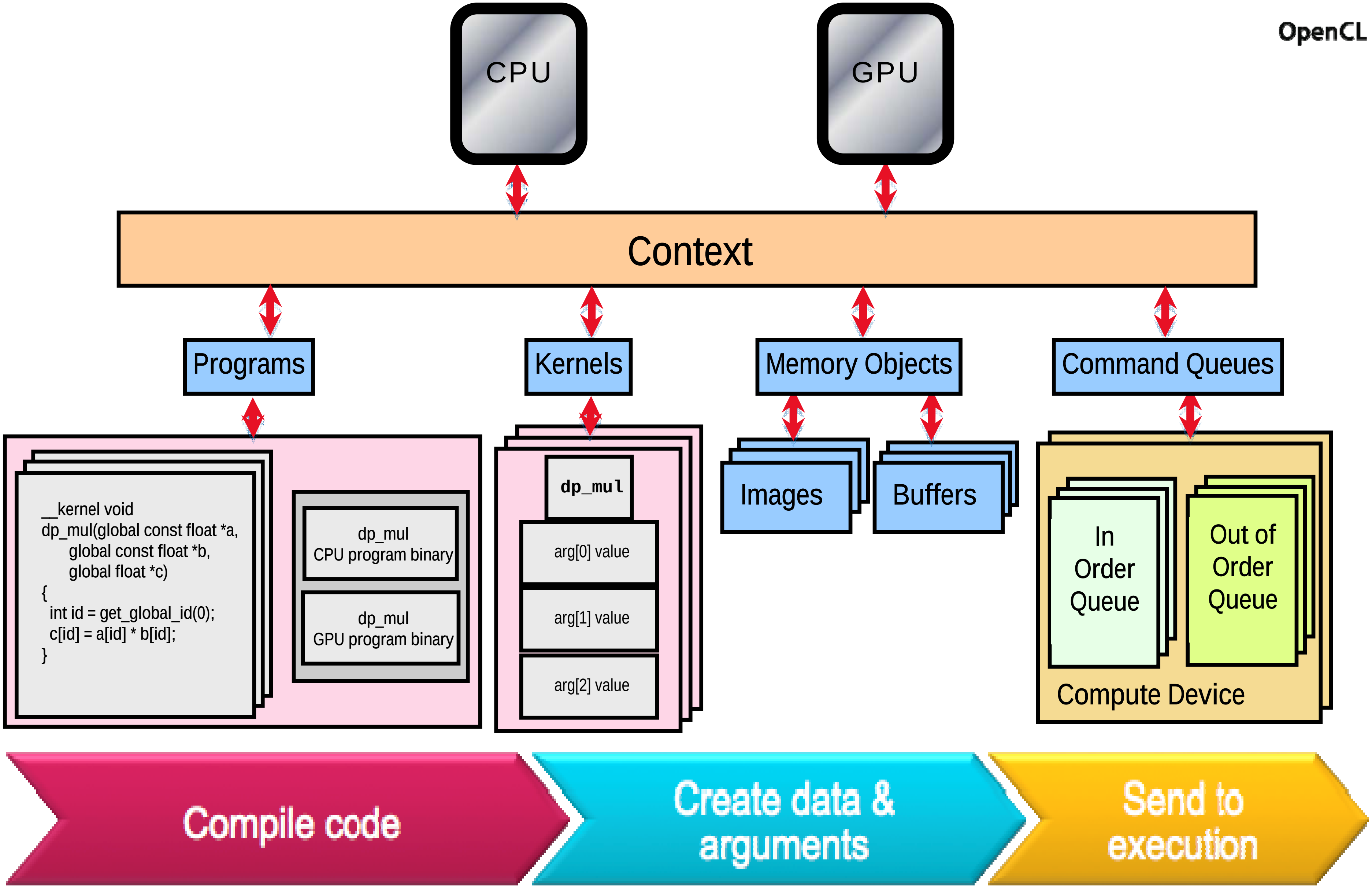
It’s complicated, but most of this is “boilerplate” and not as bad as it looks.

OpenCL Synchronization: Queues & Events

- Events can be used to synchronize kernel executions betwee queues
- Example: 2 queues with 2 devices



OpenCL summary



IS OpenCL the solution to our parallel programming problems?

- NO ... OpenCL is ugly
  - Its great for expert programmers mapping software onto low level hardware features.
  - Its great for programmers who want full control over the hardware.
  - Its terrible for end-user or domain expert programmers

We need to develop parallel programming technologies that will change the world and make *<u>every</u>* programmer a parallel programmer.

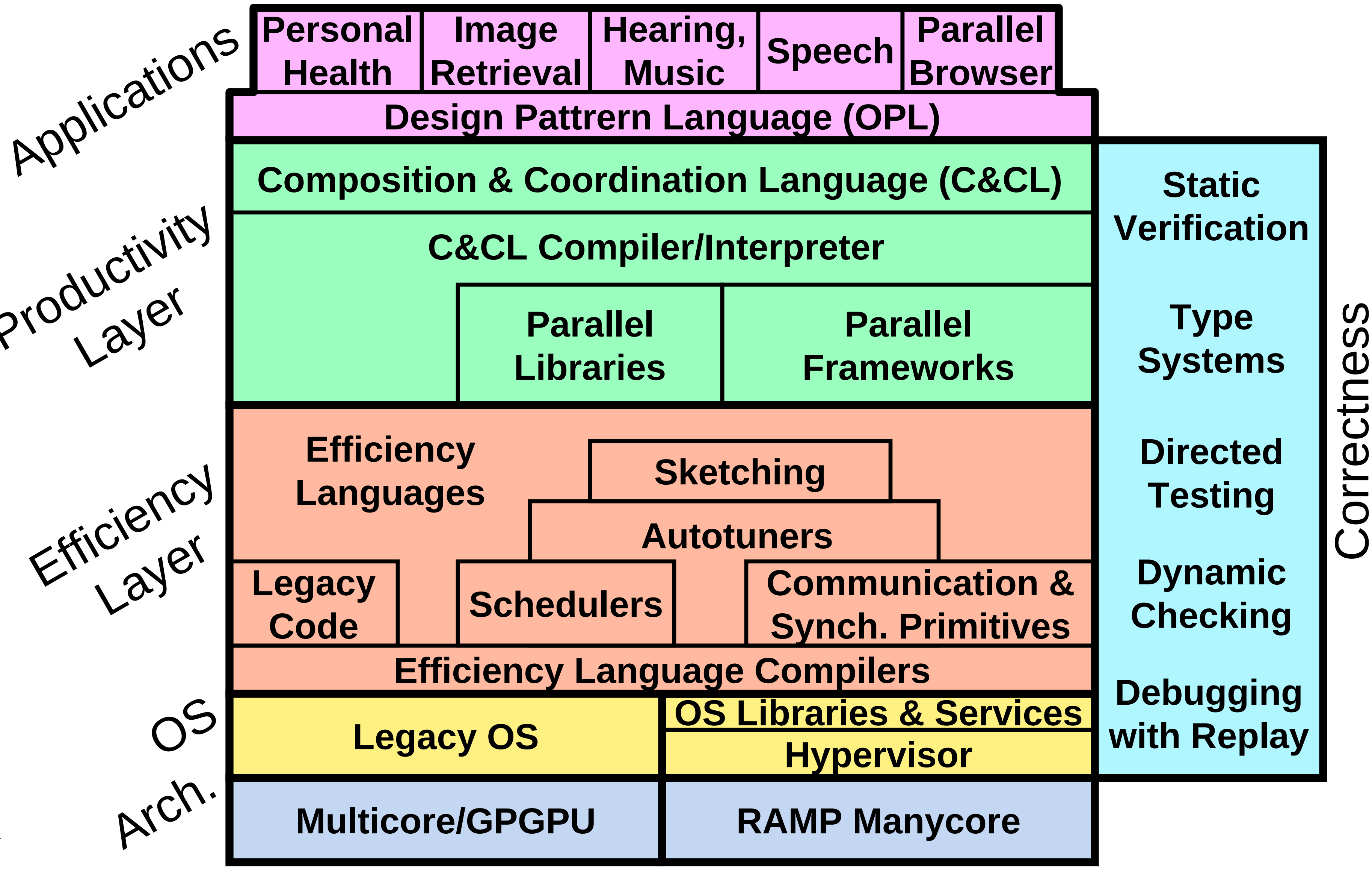
Agenda

- The many core software challenge
- OpenCL: a brief overview
- Going beyond OpenCL

UC Berkeley's Par Lab Agenda

with lots of help from Intel, Microsoft, and others

Easy to write correct programs that run efficiently on manycore

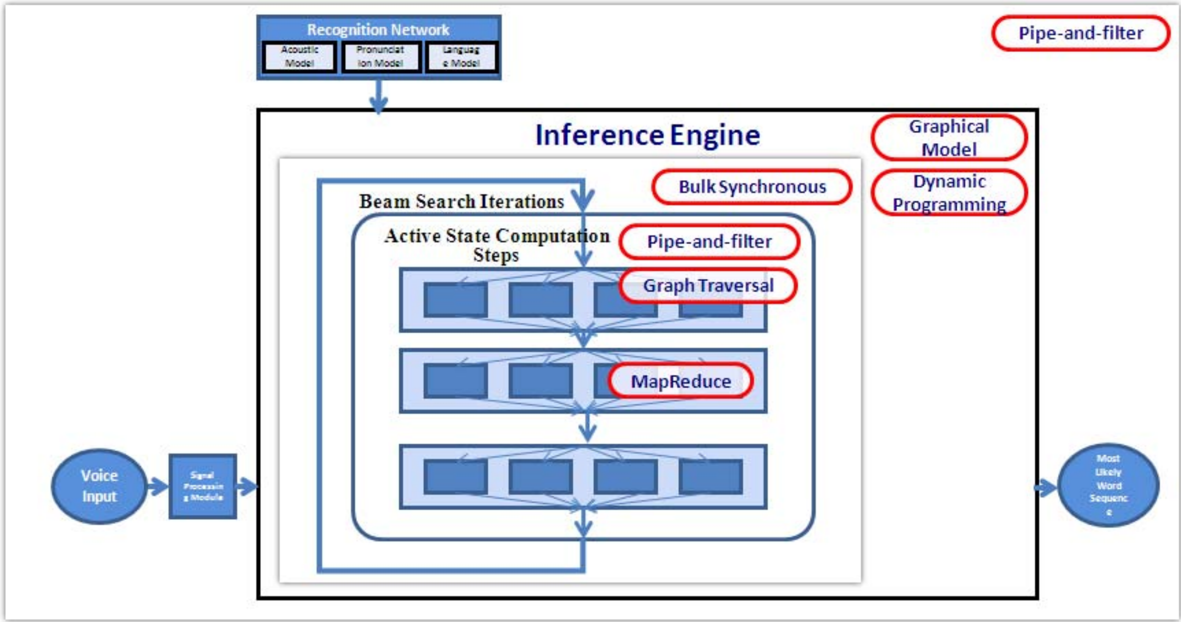


Professor Keutzer's experiences as CTO of Synopsis

- SW architecture
  - Is drastically more important than programming environments
  - which is more important than programming languages
  - Which is more important than compilers and debuggers
  - Which is farm more important than hardware.
- Super-programmer:
  - 10X productivity
  - 10X speed of code
  - 1/10th as many bugs
- Interview after interview with Synopsis' super programmers indicated ...

"give me the right architecture and you can keep your cool languages and tools – but give me a bad architecture and no amount of tools and languages will help."

Software architecture is the key

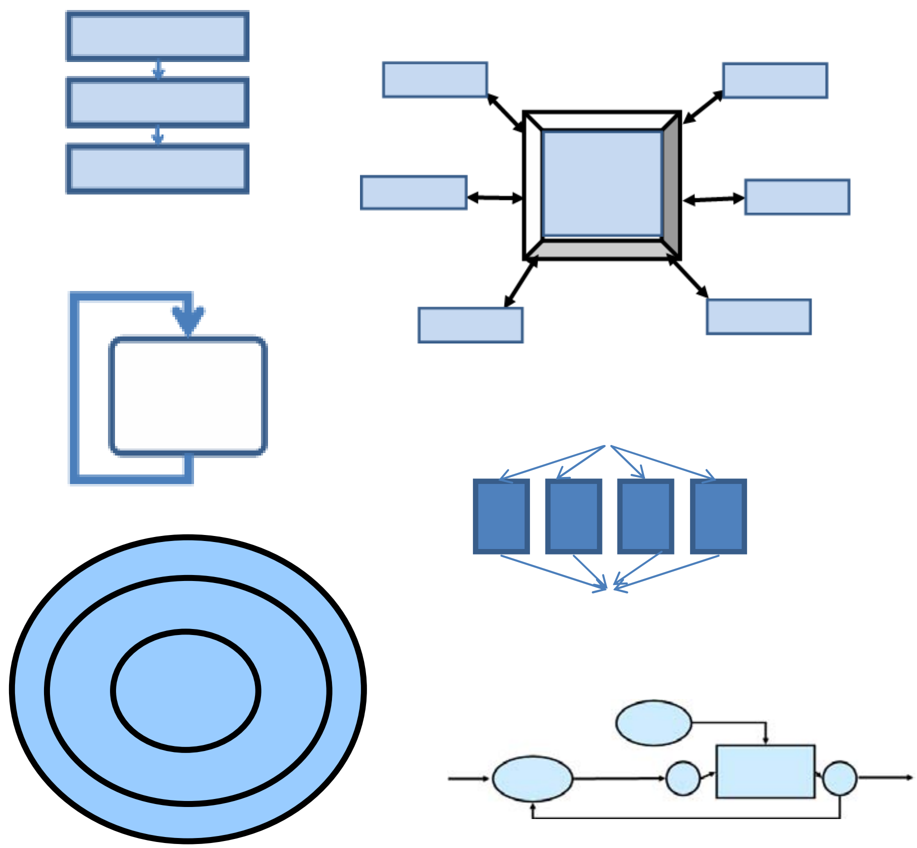


- Enforces modularity
  - Good for design, management, and performance optimization.
- Clarifies interfaces
  - Good for design, management, and debuggability
- Eases communication of design – benefits documentation and future maintenance

We use Design Patterns to write down architectural ideas

Garlan and Shaw Architectural Styles

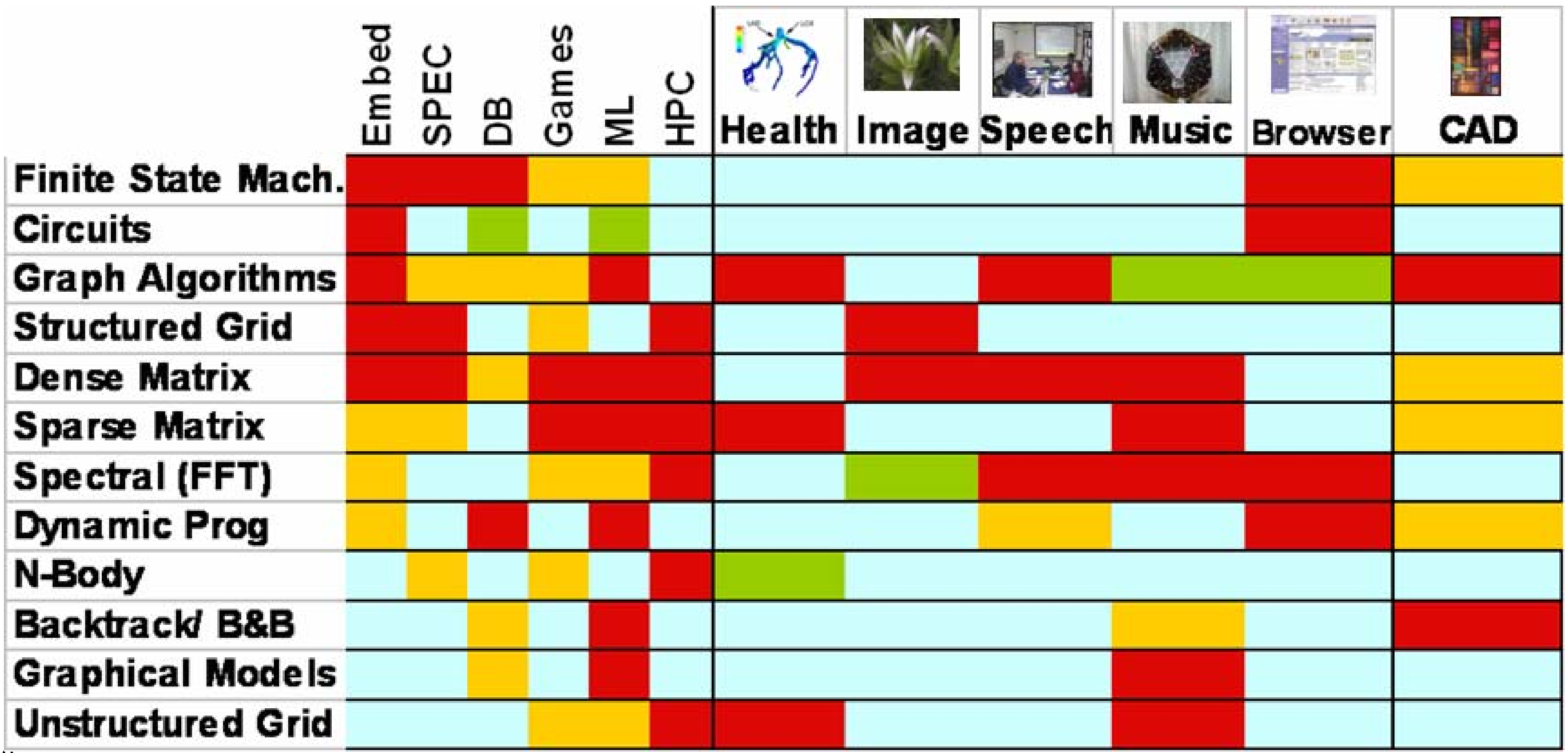
- Pipe-and-Filter
- Agent-and-Repository
- Event-based coordination
- Process Control
- Layered Systems



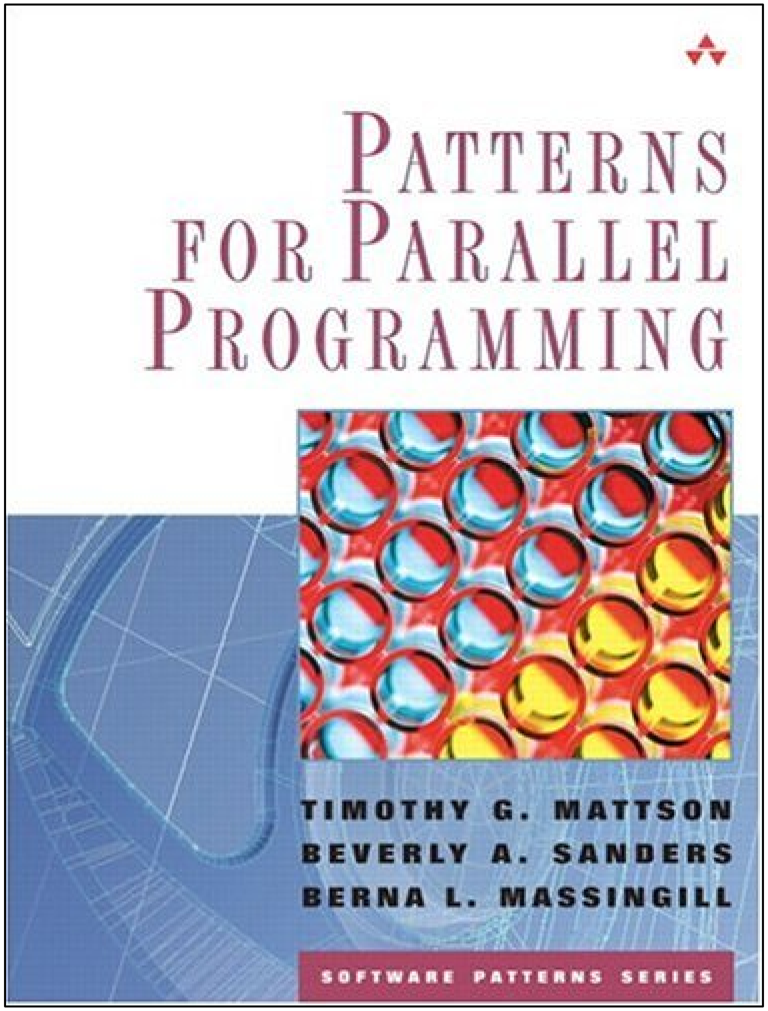
These define the structure of our software but they *do not describe* what is computed

Computational Patterns

The Dwarfs from "The Berkeley View" (Asanovic et al.)
Dwarfs form our key computational patterns



Parallel algorithm patterns



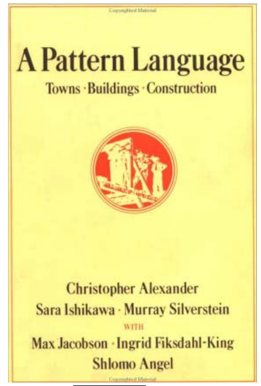
A design pattern language for parallel algorithm design with examples in MPI, OpenMP and Java.

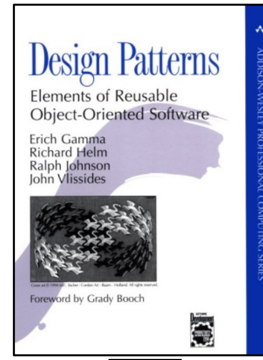
This is our hypothesis for how programmers think about parallel programming.

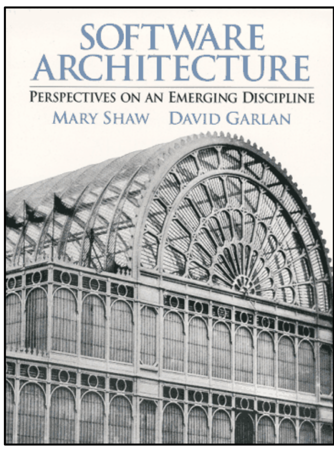
We call this PLPP (Pattern Language of Parallel Programming)

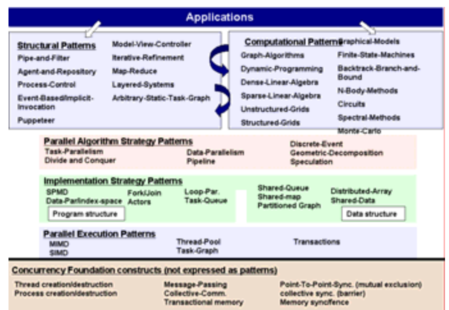
Work started in 1997.
Book published 2004

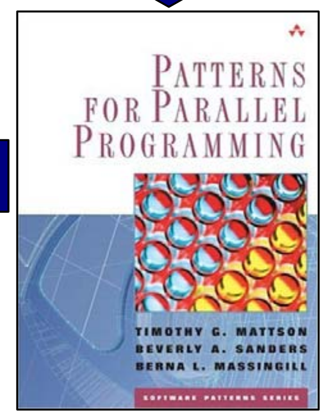
Influences on the Berkeley Pattern language











13 dwarves

OPL 2010 (Keutzer and Mattson Intel Technology Journal, 2010)

Applications

Structural Patterns

- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Puppeteer
- Model-View-Controller
- Iterative-Refinement
- Map-Reduce
- Layered-Systems
- Arbitrary-Static-Task-Graph

Computational Patterns

- Graph-Algorithms
- Dynamic-Programming
- Dense-Linear-Algebra
- Sparse-Linear-Algebra
- Unstructured-Grids
- Structured-Grids
- Graphical-Models
- Finite-State-Machines
- Backtrack-Branch-and-Bound
- N-Body-Methods
- Circuits
- Spectral-Methods
- Monte-Carlo

Parallel Algorithm Strategy Patterns

- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline
- Discrete-Event
- Geometric-Decomposition
- Speculation

Implementation Strategy Patterns

- SPMD
- Data-Par/index-space
- Fork/Join
- Actors
- Loop-Par.
- Task-Queue

Program structure

- Shared-Queue
- Shared-map
- Partitioned Graph
- Distributed-Array
- Shared-Data

Data structure

Parallel Execution Patterns

- MIMD
- SIMD
- Thread-Pool
- Task-Graph
- Transactions

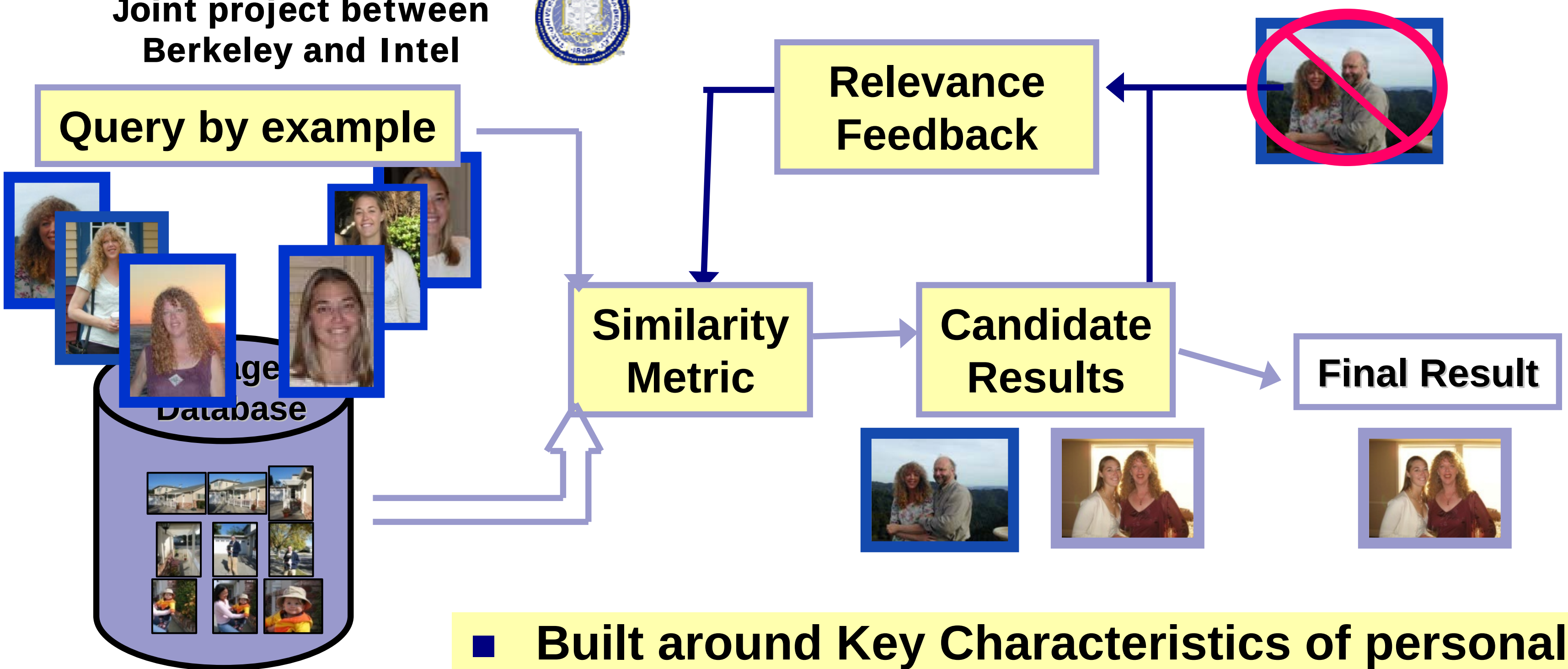
Concurrency Foundation constructs (not expressed as patterns)

- Thread creation/destruction
- Process creation/destruction
- Message-Passing
- Collective-Comm.
- Transactional memory
- Point-To-Point-Sync. (mutual exclusion)
- collective sync. (barrier)
- Memory sync/fence

Content-Based Image Retrieval Overview

piro

Piro: Personal Image Retrieval Organizer
Joint project between Berkeley and Intel

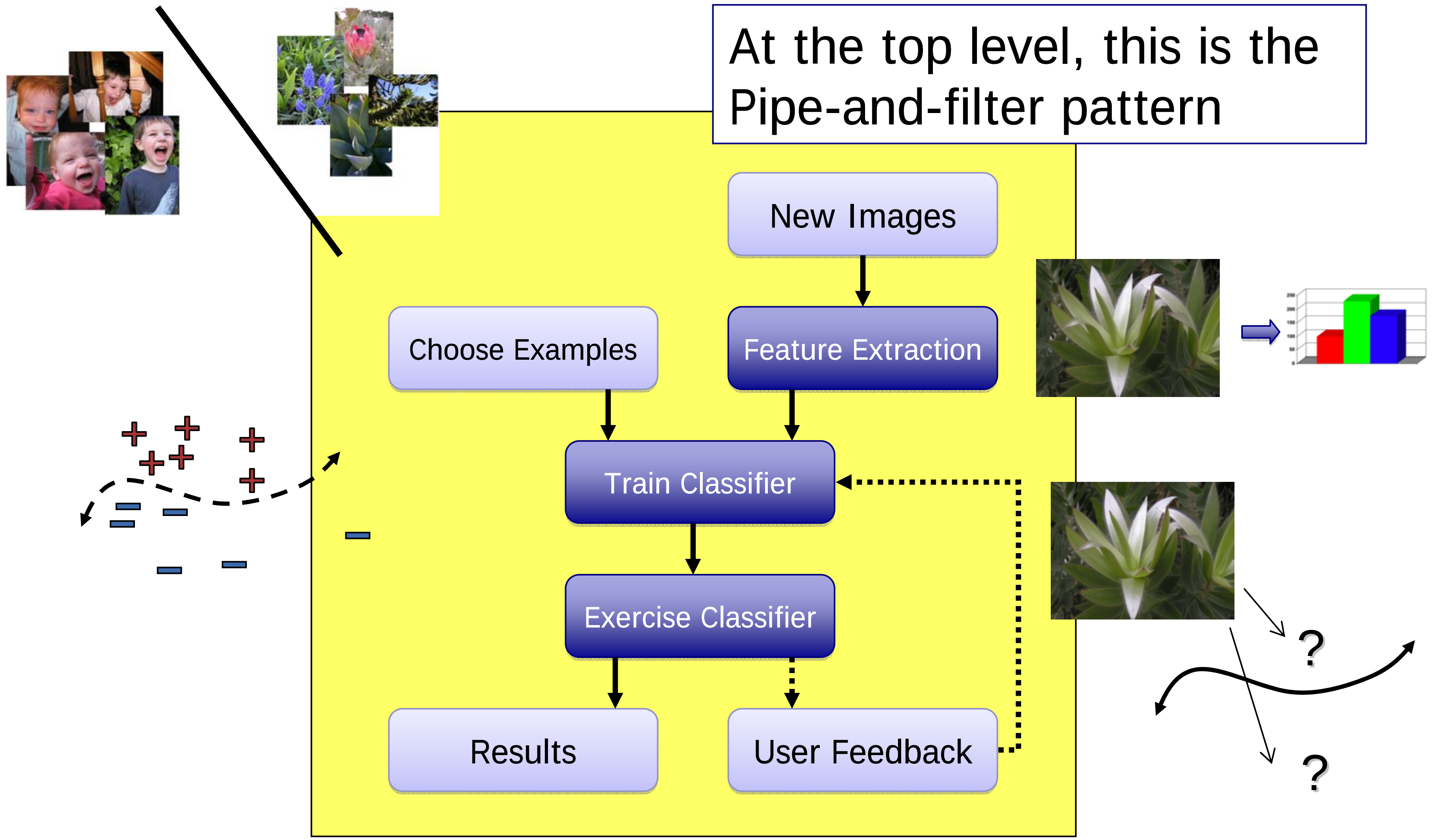


1000's → 1M images

- **Built around Key Characteristics of personal databases**
  - **Very large number of pictures (1K → 1M)**
  - **Non labeled images**
  - **Many pictures of few people**
  - **Complex pictures including people, events, places, and objects**

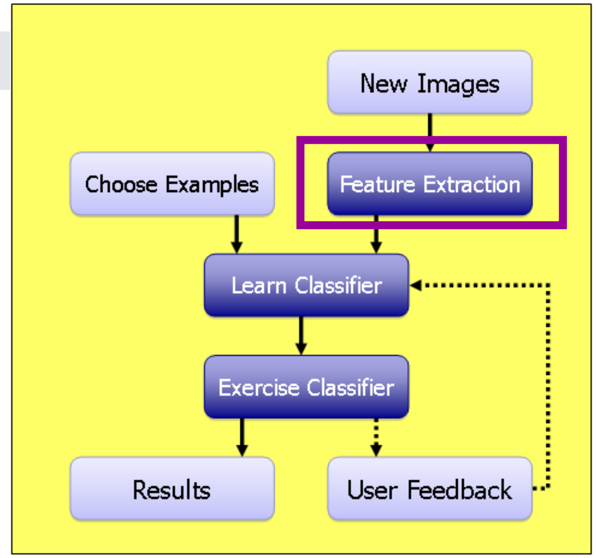
CBIR Application Framework

At the top level, this is the Pipe-and-filter pattern

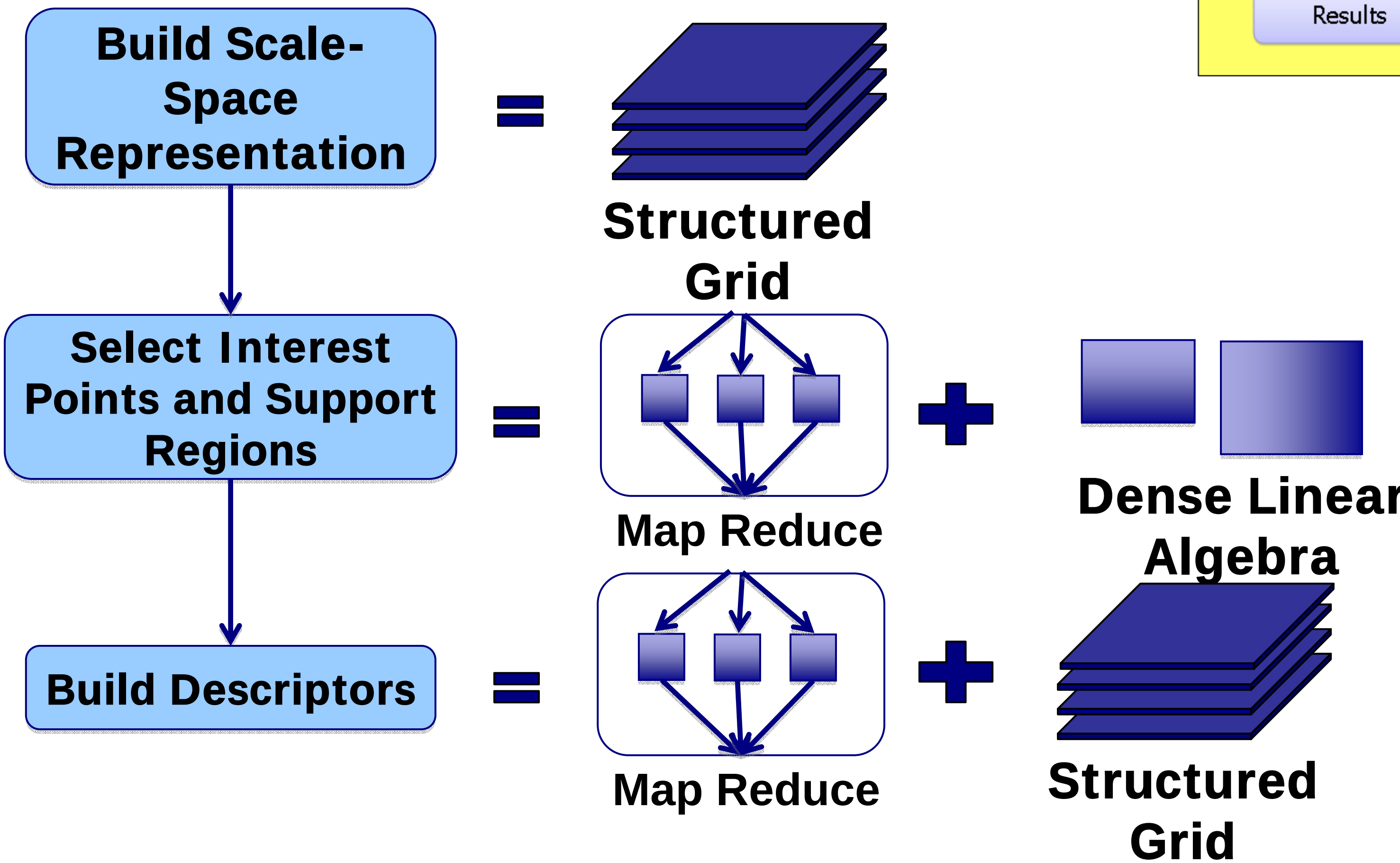


Catanzaro, Sundaram, Keutzer, "Fast SVM Training and Classification on Graphics Processors", Int'l Conf. Machine Learning 2008

Feature Extraction

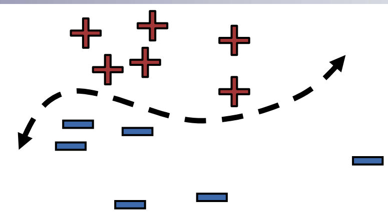


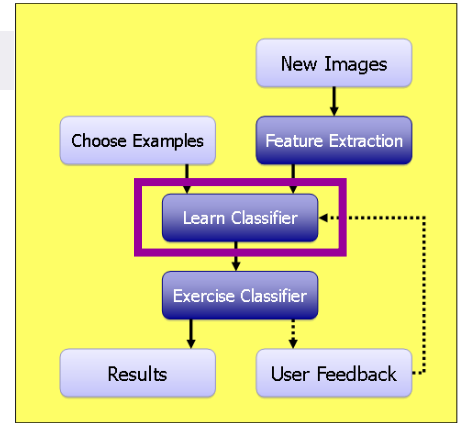
- Image is reduced to a set of low-dimensional feature vectors
-

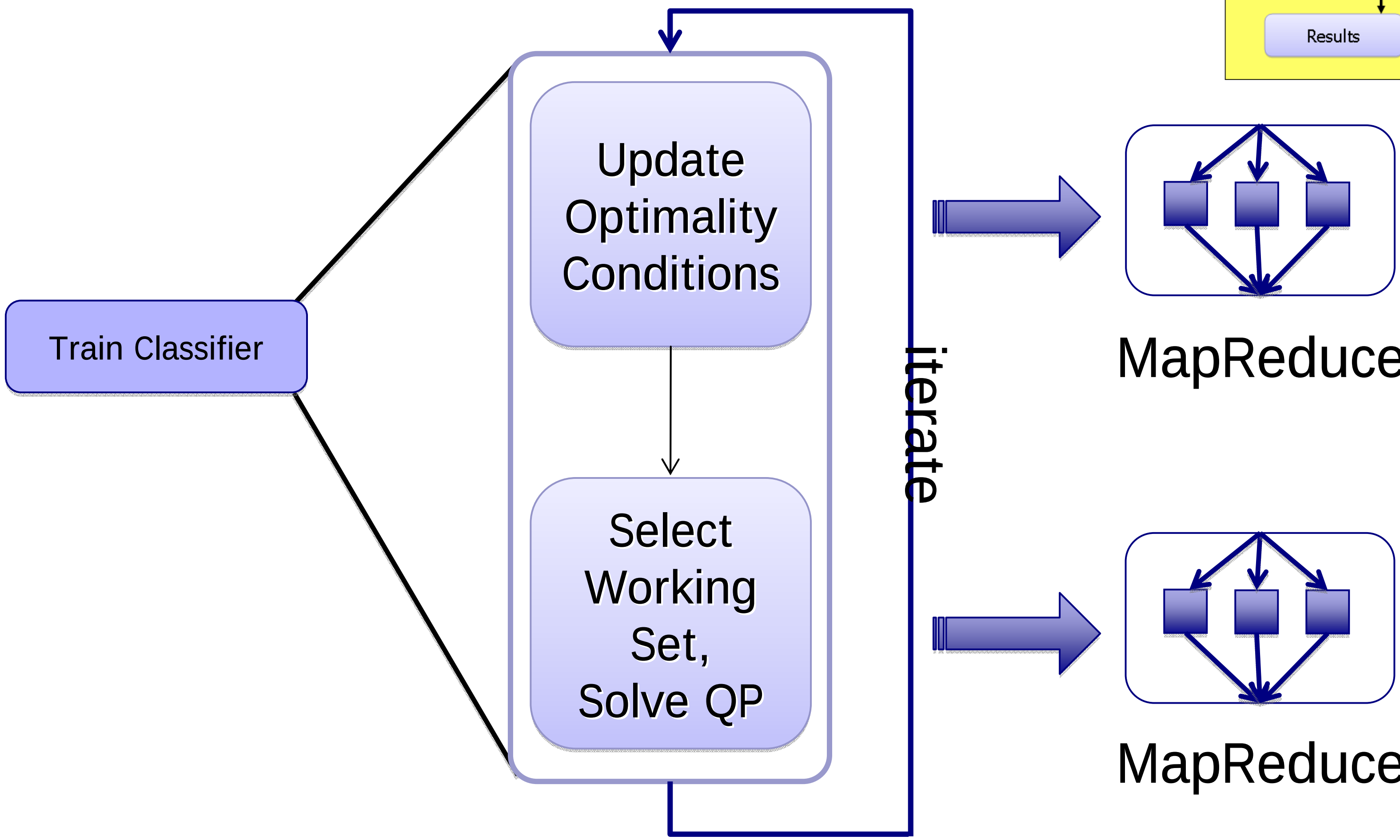


"Image Feature Extraction for Mobile Processors", Mark Murphy, Hong Wang, Kurt Keutzer IISWC '09

Train Classifier: SVM Training

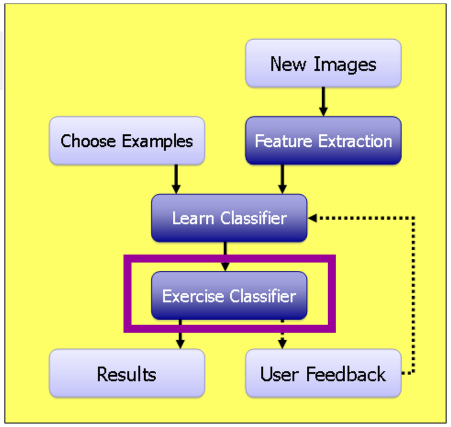


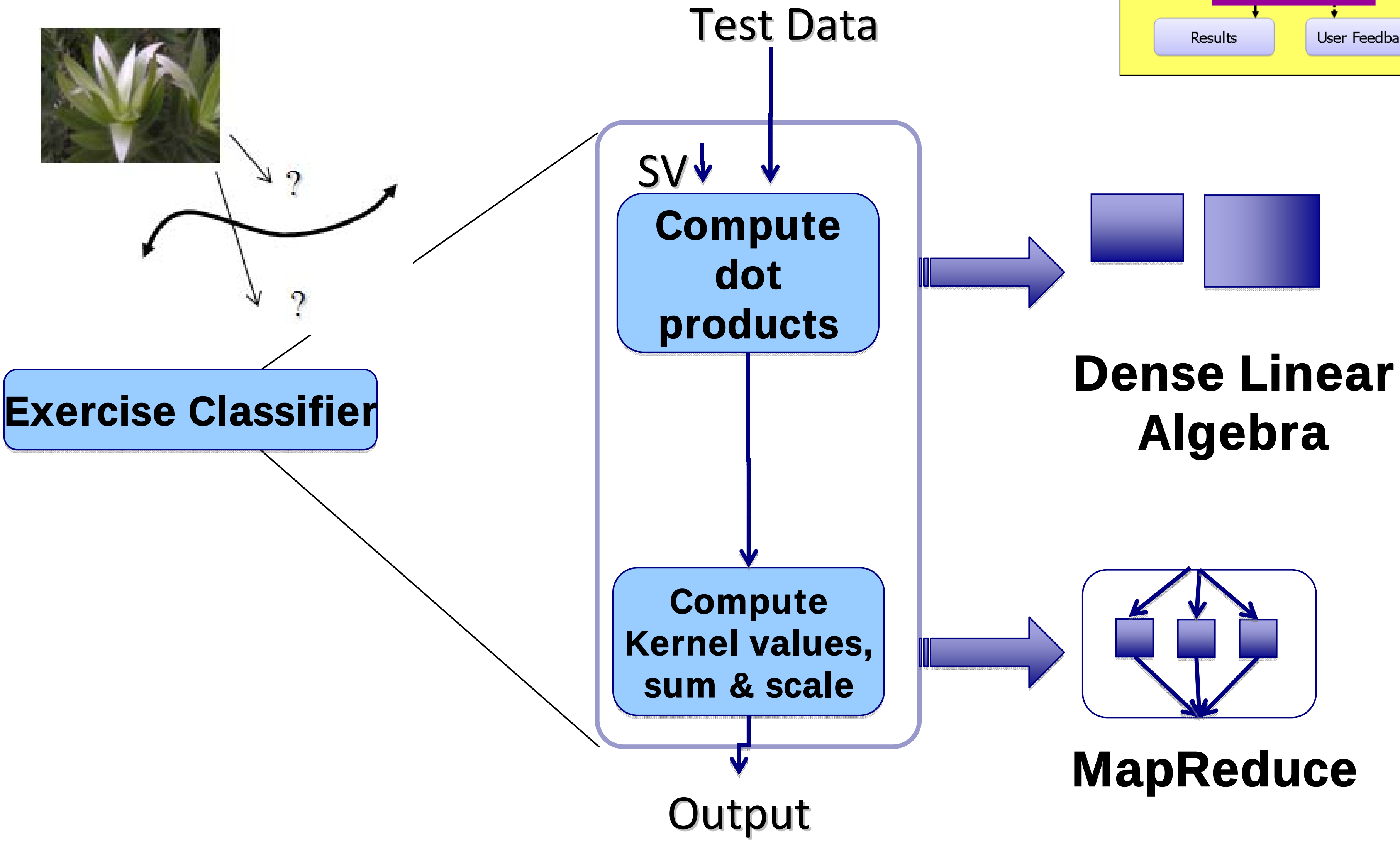




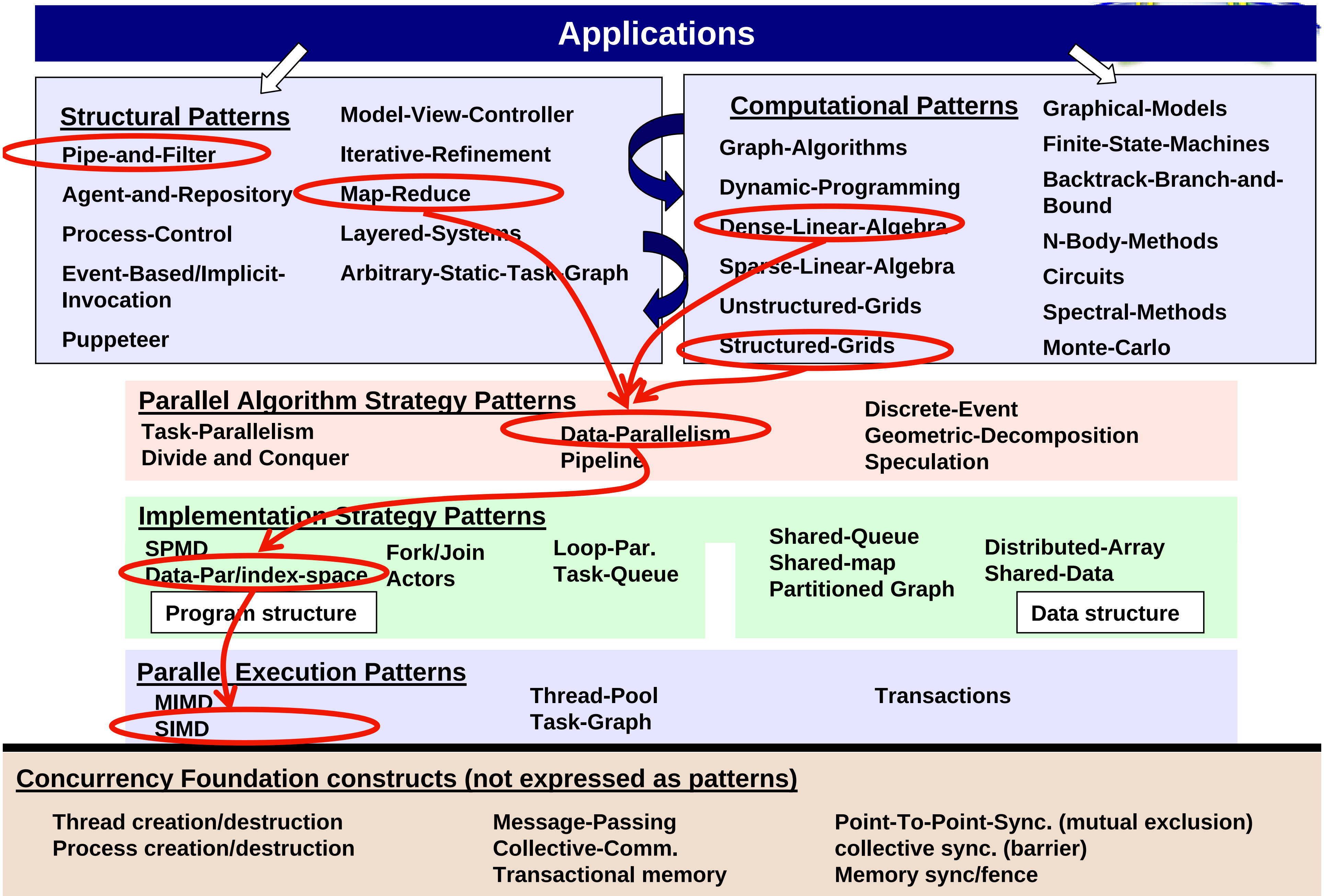
Iterative Refinement Structural Pattern

Exercise Classifier : SVM Classification





Patterns travel together … and this informs framework design



Turning Patterns into code

- Frameworks:
  - Raise the level of abstraction .. help turn patterns into code.
  - Support a separation of concern … concurrency-experts build the frameworks, domain programmers just use them.

Example: copperhead, a framework for writing data parallel code with python. Maps onto CUDA today, OpenCL work in progress

- Consider this intrinsically parallel procedure

```
def saxpy(a, x, y):
  return map(lambda xi,yi: a*xi + yi,  x,  y)
```

… or for the lambda averse …

```
def saxpy(a, x, y):
  return [a*xi + yi for xi,yi in zip(x,y)]
```

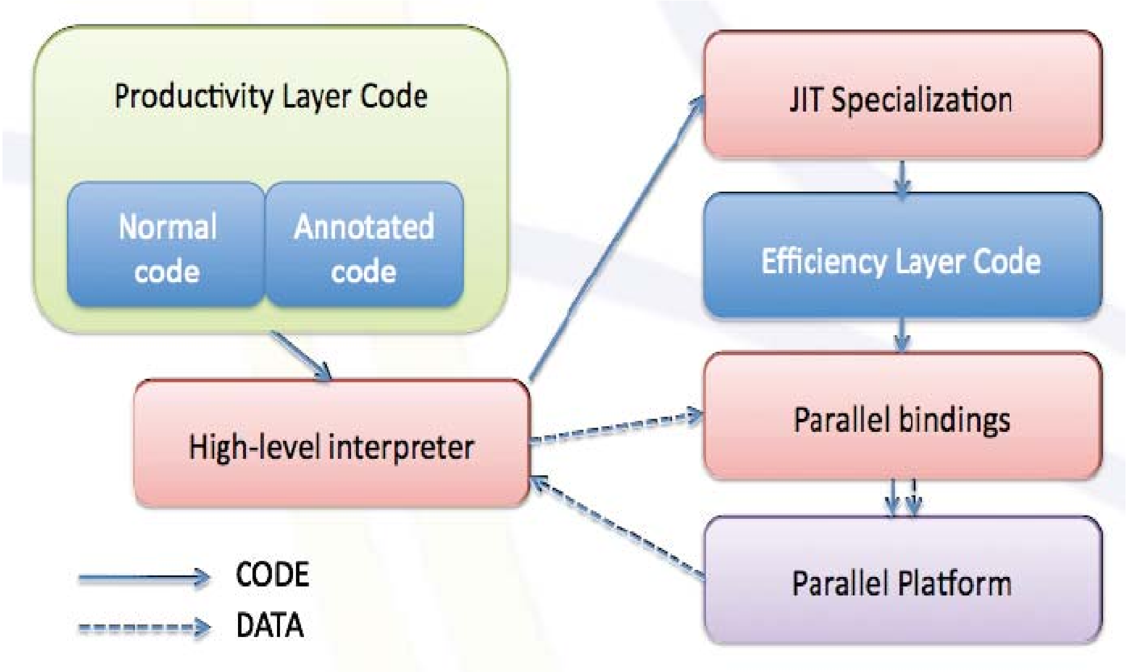
- This procedure is both
  - completely valid Python code
  - compilable to data parallel languages like CUDA or OpenCL

Bryan Catanzaro UC Berkely, Michael Garland, Nvidia

But what about performance?

- SEJITS: Scalable, embedded, just in time specialization
  - Write python annotated for data parallel programming.
  - SEJITS system to embed optimized kernels specialized at runtime to flatten abstraction overhead and map onto hardware features.

Currently works with copperhead and Ruby→C/OpenMP



Bryan Catanzaro, Armando fox, Yunsup Lee, mark Murphy and Kurt Ketuzer of UC Berkeley, Mickael Garland of NVIDIA

Summary

- Many core chips are coming … 100’s or even 1000 cores over the next 15 years.
- SW is not ready for these chips.
- The key is standards … and they will not become established if programmers don’t demand them.
- OpenCL is a new standard for programming the heterogeneous platform.
- But OpenCL is not enough … it addresses the needs of “efficiency layer” programmers. We need something more to address more typical “productivity layer” programmers.
- We are mid-way through a research collaboration with UC Berkeley to address this:
  - Design patterns to guide our solutions
  - Frameworks to make common patterns easy to code
  - Embedded, dynamic specialization for performance

Backup

- A simple example

Example: vector addition

- The “hello world” program of data parallel programming is a program to add two vectors

C[i] = A[i] + B[i] for i=1 to N

- For the OpenCl solution, there are two parts
  - Kernel code
  - Host code

Platform Layer: Basic discovery

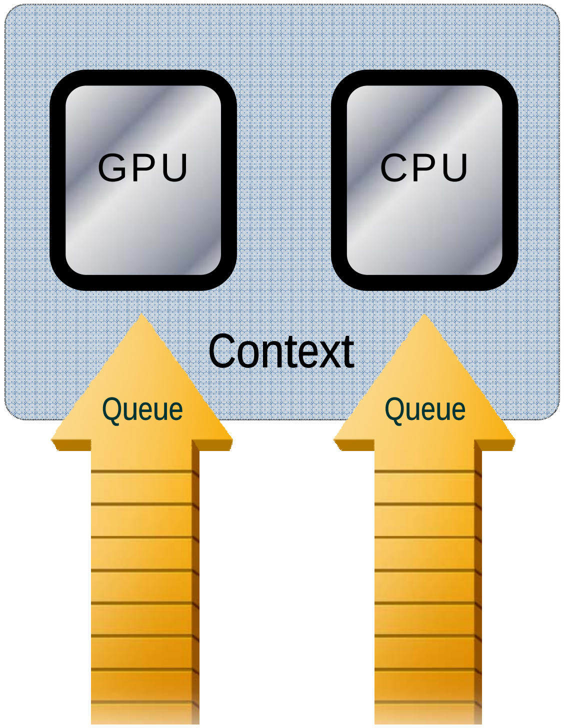
- Platform layer allows applications to query for platform specific features
- Querying platform info Querying devices
  - *clGetDeviceIDs*()
    - Find out what compute devices are on the system
    - Device types include CPUs, GPUs, or Accelerators
  - *clGetDeviceInfo*()
    - Queries the capabilities of the discovered compute devices such as:
      - Number of compute cores
      - Maximum work-item and work-group size
      - Sizes of the different memory spaces
      - Maximum memory object size

Platform Layer: Contexts

- Creating contexts
  - Contexts are used by the OpenCL runtime to manage objects and execute kernels on one or more devices
  - Contexts are associated to one or more devices
    - Multiple contexts could be associated to the same device
  - *clCreateContext*() and *clCreateContextFromType*() returns a *handle* to the created contexts

Platform layer: Command-Queues

- Command-queues store a set of operations to perform
- Command-queues are associated to a context
- Multiple command-queues can be created to handle independent commands that don't require synchronization
- Execution of the command-queue is guaranteed to be completed at sync points



VecAdd: Context, Devices, Queue

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(
             0,                    // platform ID
             CL_DEVICE_TYPE_GPU,   // Ask for a GPU
             NULL,                 // error callback
             NULL,                 // user data for callback
             NULL);                // error code

// get the list of GPU devices associated with context
size_t cb;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
                                           devices, NULL);

// create a command-queue
cl_cmd_queue cmd_queue = clCreateCommandQueue(context,
             devices[0], // Use the first GPU device
             0,          // default options
             NULL);      // error code
```

Memory Objects

- Buffers
  - Simple chunks of memory
  - Kernels can access however they like (array, pointers, structs)
  - Kernels can read and write buffers
- Images
  - Opaque 2D or 3D formatted data structures
  - Kernels access only via read_image() and write_image()
  - Each image can be read or written in a kernel, but not both

Creating Memory Objects

- Memory objects are created within an associated context
  - *clCreateBuffer()*, *clCreateImage2D*(), and *clCreateImage3D*()
- Memory can be created as read only, write only, or read-write
- Where objects are created in the platform memory space can be controlled
  - Device memory
  - Device memory with data copied from a host pointer
  - Host memory
  - Host memory associated with a pointer
    - Memory at that pointer is guaranteed to be valid at synchronization points

VecAdd: Create Memory Objects

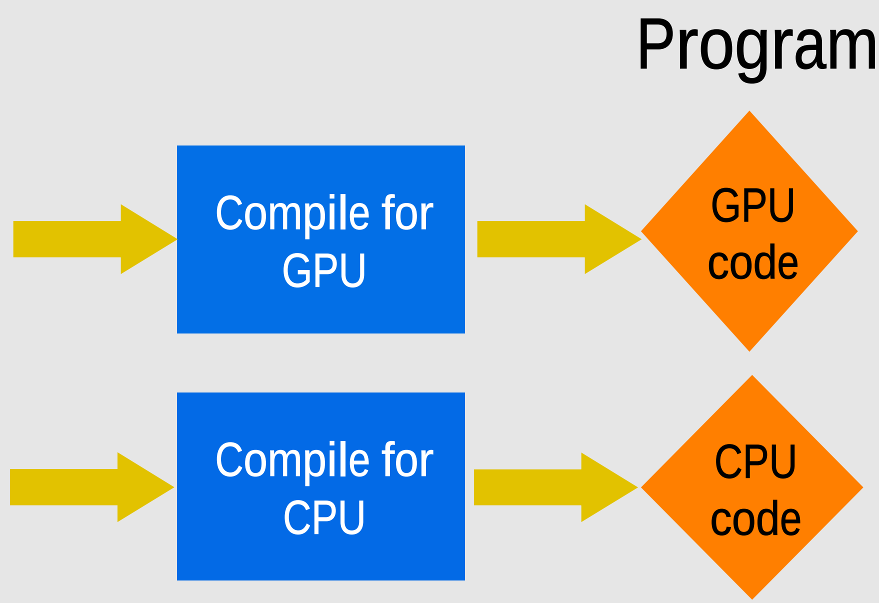
```
cl_mem memobjs[3];
// allocate input buffer memory objects
memobjs[0] = clCreateBuffer(context,
                   CL_MEM_READ_ONLY |    // flags
                   CL_MEM_COPY_HOST_PTR,
                   sizeof(cl_float)*n,   // size
                   srcA,                 // host pointer
                   NULL);                // error code
memobjs[1] = clCreateBuffer(context,
            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
            sizeof(cl_float)*n, srcB, NULL);

// allocate output buffer memory object
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);
```

Build the Program object

- The program object encapsulates:
  - A context
  - The program source/binary
  - List of target devices and build options
- The Build process … to create a program object
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

Kernel Code

```
kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                        (int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



VecAdd: Create and Build the Program

```
// create the program
cl_program program = clCreateProgramWithSource(
                context,
                1,                // string count
                &program_source, // program strings
                NULL,             // string lengths
                NULL);            // error code
// build the program
cl_int err = clBuildProgram(program,
                0,      // device num within the device list
                NULL,   // device list
                NULL,   // options
                NULL,   // notifier callback function ptr
                NULL); // user data for callback function
```

Kernel Objects

- Kernel objects encapsulate
  - Specific kernel functions declared in a program
  - Argument values used for kernel execution
- Creating kernel objects
  - *clCreateKernel*() - creates a kernel object for a single function in a program
- Setting arguments
  - *clSetKernelArg*(<kernel>, <argument index>)
  - Each argument data must be set for the kernel function
  - Argument values copied and stored in the kernel object
- Kernel vs. program objects
  - Kernels are related to program execution
  - Programs are related to program source

VecAdd: Create the Kernel and Set the Arguments

```
// create the kernel
cl_kernel kernel = clCreateKernel(program, “vec_add”, NULL);

// set “a” vector argument
err  = clSetKernelArg(kernel,
               0,                    // argument index
               (void *)&memobjs[0], // argument data
               sizeof(cl_mem));     // argument data size

// set “b” vector argument
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                sizeof(cl_mem));

// set “c” vector argument
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                sizeof(cl_mem));
```

Kernel Execution

- A command to execute a kernel must be enqueued to the command-queue
  - Command-queue could be explicitly flushed to the device
  - Command-queues execute in-order or out-of-order
    - In-order - commands complete in the order queued and memory is consistent
    - Out-of-order - no guarantee of (1) when commands are executed or (2) if memory is consistent … unless specific synchronization is used.
- *clEnqueueNDRangeKernel*()
  - Data-parallel execution model
  - Describes the ***index space*** for kernel execution
  - Requires information on NDRange dimensions and work-group size
- *clEnqueueTask*()
  - Task-parallel execution model (multiple queued tasks)
  - Kernel is executed on a single work-item
- *clEnqueueNativeKernel*()
  - Task-parallel execution model
  - Executes a native C/C++ function not compiled using the OpenCL compiler
  - This mode does not use a kernel object so arguments must be passed in

VecAdd: Invoke Kernel

```
size_t global_work_size[1] = n; // set work-item dimensions
// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel,
                    1,          // Work dimensions
                    NULL,       // must be NULL (work offset)
                    global_work_size,
                    NULL,       // automatic local work size
                    0,          // no events to wait on
                    NULL,       // event list
                    NULL);      // event for this kernel
```

Synchronization

- Synchronization
  - Signals when commands are completed to the host or other commands in queue
  - Blocking calls
    - Commands that do not return until complete
    - clEnqueueReadBuffer() can be called as blocking and will block until complete
  - *Event objects*
    - Tracks execution status of a command
    - Some commands can be blocked until event objects signal a completion of previous command
      - clEnqueueNDRangeKernel() can take an event object as an argument and wait until a previous command (e.g., clEnqueueWriteBuffer) is complete
  - Queue barriers - queued commands that can block command execution

VecAdd: Read Output

```
// read output array
err = clEnqueueReadBuffer( context, memobjs[2],
                    CL_TRUE,            // blocking
                    0,                  // offset
                    n*sizeof(cl_float), // size
                    dst,                // pointer
                    0, NULL, NULL);     // events
```

OpenCL C for Compute Kernels

- Derived from ISO C99
  - A few restrictions: recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported
- Built-in Data Types
  - Scalar and vector data types, Pointers
  - Data-type conversion functions: convert_type<_sat><_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t
- Built-in Functions — Required
  - work-item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
- Built-in Functions — Optional
  - double precision, atomics to global and local memory
  - selection of rounding mode, writes to image3d_t surface

Vector Addition Kernel

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
 }
```