What have we learned about programming heterogeneous computing systems?

Wen-mei Hwu

Professor and Sanders-AMD Chair, ECE, NCSA

University of Illinois at Urbana-Champaign

With

Liwen Chang, Simon Garcia, Abdul Dakkak, Hee-Seok Kim, Izzat El Hajj

ECE ILLINOIS

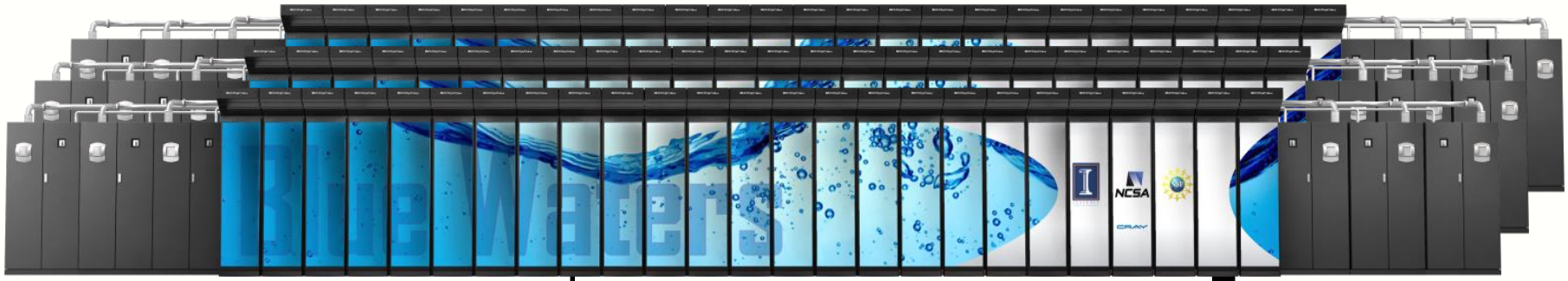
The IMPACT Research Group

NCSA

ILLINOIS

Blue Waters Computing System

Operational at Illinois since 3/2013



12.5 PF
1.6 PB DRAM
$250M

10/40/100 Gb Ethernet Switch

IB Switch

>1 TB/sec

120+ Gb/sec

100 GB/sec

WAN

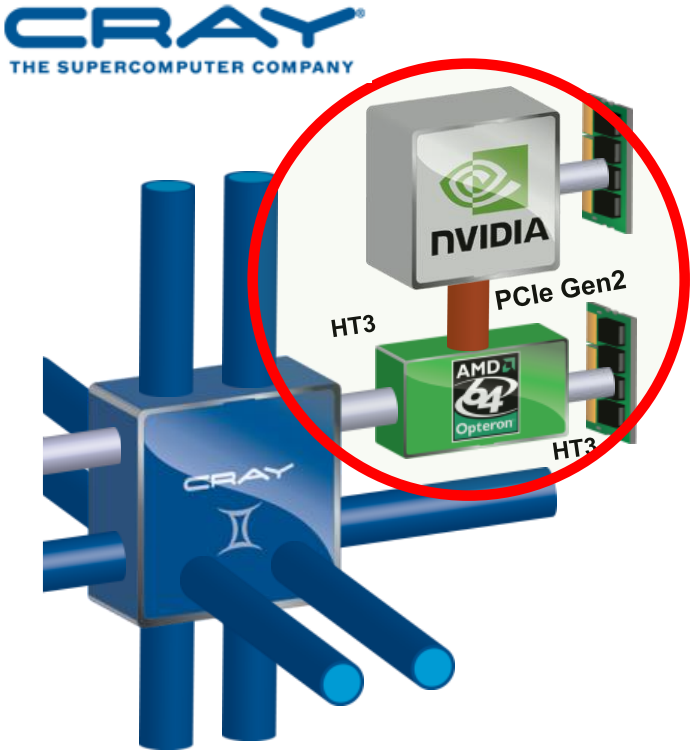
Spectra Logic: 300 PBs

Sonexion: 26 PBs

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Heterogeneous Computing in Blue Waters

- Dual-socket Node
  - One AMD Interlagos chip
    - 8 core modules, 32 threads
    - 156.5 GFs peak performan
    - 32 GBs memory
      - 51 GB/s bandwidth
  - One NVIDIA Kepler chip
    - 1.3 TFs peak performance
    - 6 GBs GDDR5 memory
      - 250 GB/sec bandwidth
  - Gemini Interconnect



Blue Waters contains 4,224 Cray XK7 compute nodes.

Initial Production Use Results

- NAMD
  - 100 million atom benchmark with Langevin dynamics and PME once every 4 steps, from launch to finish, all I/O included
  - 768 nodes, Kepler+Interlagos is 3.9X faster over Interlagos-only
  - 768 nodes, XK7 is 1.8X XE6
- Chroma
  - Lattice QCD parameters: grid size of 483 x 512 running at the physical values of the quark masses
  - 768 nodes, Kepler+Interlagos is 4.9X faster over Interlagos-only
  - 768 nodes, XK7 is 2.4X XE6
- QMCPACK
  - Full run Graphite 4x4x1 (256 electrons), QMC followed by VMC
  - 700 nodes, Kepler+Interlagos is 4.9X faster over Interlagos-only
  - 700 nodes, XK7 is 2.7X XE6

Some Lessons Learned

- Throughput computing using GPUs can result in 2-3X end-to-end application level performance improvement
- GPU computing has had narrow but deep impact in the application space due to limited support for CPU-GPU collaboration
  - Small GPU memory and data movement overhead
  - Coarse grained platform-level workflow
  - **Low-level programming interfaces with poor performance portability**

Performance Library

- A major qualifying factor for new computing platforms
- Currently redeveloped and hand-tuned for each HW type/generation
- Exa-scale HW expected to have increasing levels of heterogeneity, parallelism, and hierarchy
  - Increasing levels of memory heterogeneity and hierarchy
  - Increase SIMD width and number of cores
- Performance library development process must keep up with the HW evolution and diversification
  - Performance portability

It is not just about supercomputing

- Smart phone computing apps
- Software defined networking
- Autonomous vehicle image analysis
- Cloud services for image search and management
- IoT devices
- …

Trend Towards Heterogeneity

1 core

2003

4 cores

2006

SoC (1 core)

2008

6 cores

2010

SoC (2 cores)

many-core

2012

SoC (6 cores)

OpenPower
CAPI

2014

2005

2 cores

2007

many-core

2010

NVIDIA
Fermi™

many-core

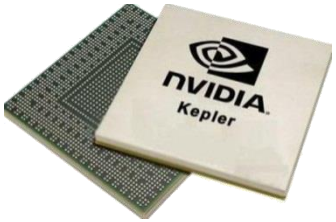
Stellarton

CPU+FPGA

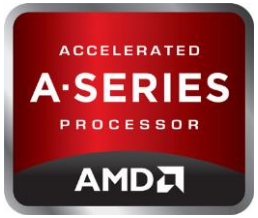
2011

APU (1st gen)

2012

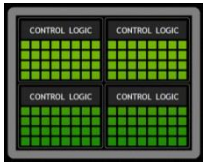


many-core



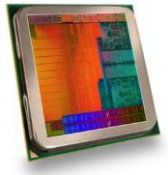
APU (2nd gen)

2014



NVIDIA
Maxwell

many-core



Kaveri

APU (3rd gen)

C++ Sequential Reduction

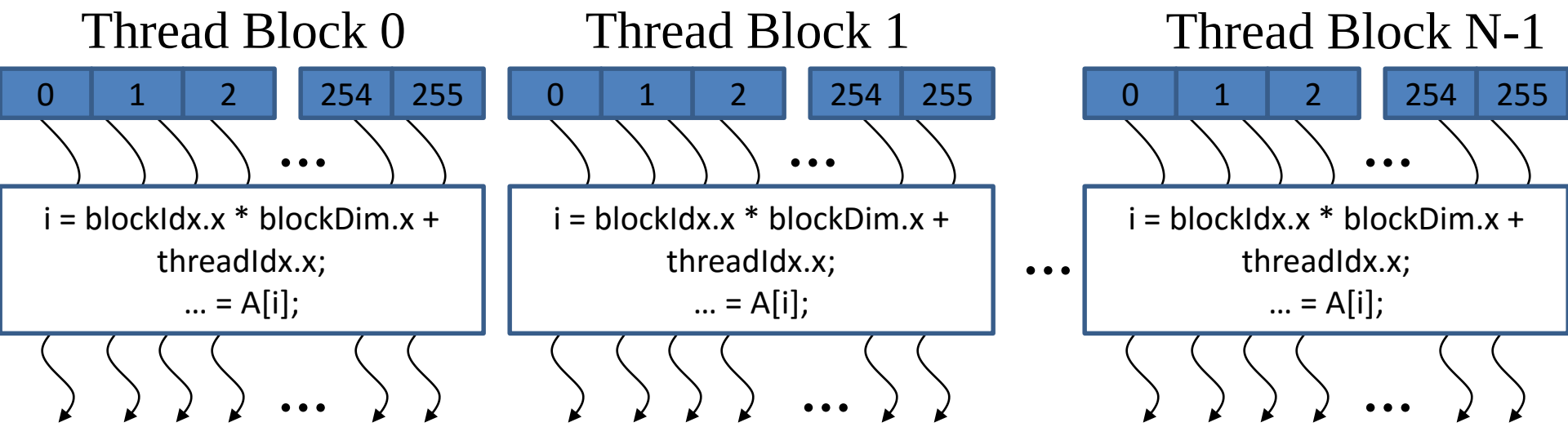
```
float reduce(const Array<1,float> in) {
  int len = in.size();
  int accum = 0;
  for(int i=0; i<len; i++) {
    accum += in[i];
  }
  return accum;
}
```

CUDA Parallel Reduction

```
__global__
void reduce(float* input, int length) {

  __shared__ float partialSum[2*BLOCK_SIZE];
  unsigned int t = threadIdx.x;
  unsigned int start = 2*blockIdx.x*blockDim.x;

  partialSum[t] = input[start + t];
  partialSum[blockDim.x+t] = input[start+blockDim.x+t];

  for (unsigned int stride = blockDim.x;
     stride > 0;  stride /= 2)
  {
    __syncthreads();
    if (t < stride)
       partialSum[t] += partialSum[t+stride];
  }
```

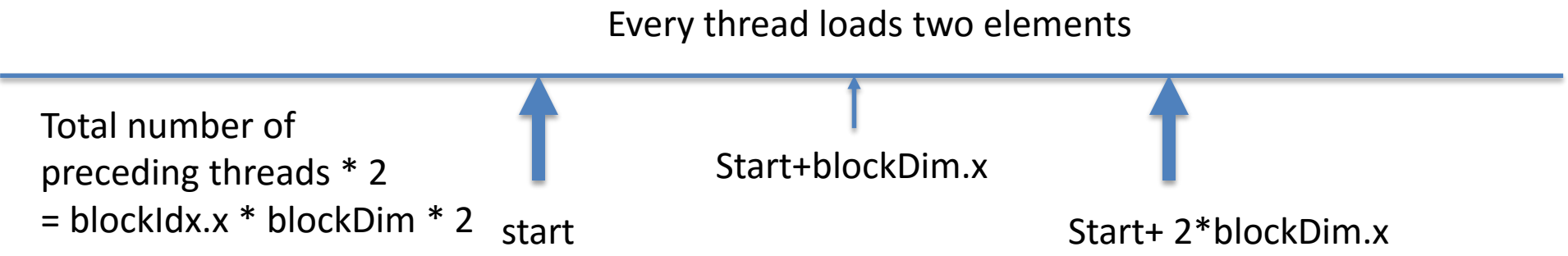
CUDA Threads and Blocks - Basics

- Divide thread array into multiple blocks
  - Threads within a block efficiently cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks do not interact
  - Threads and Blocks have unique indices for data access mapping



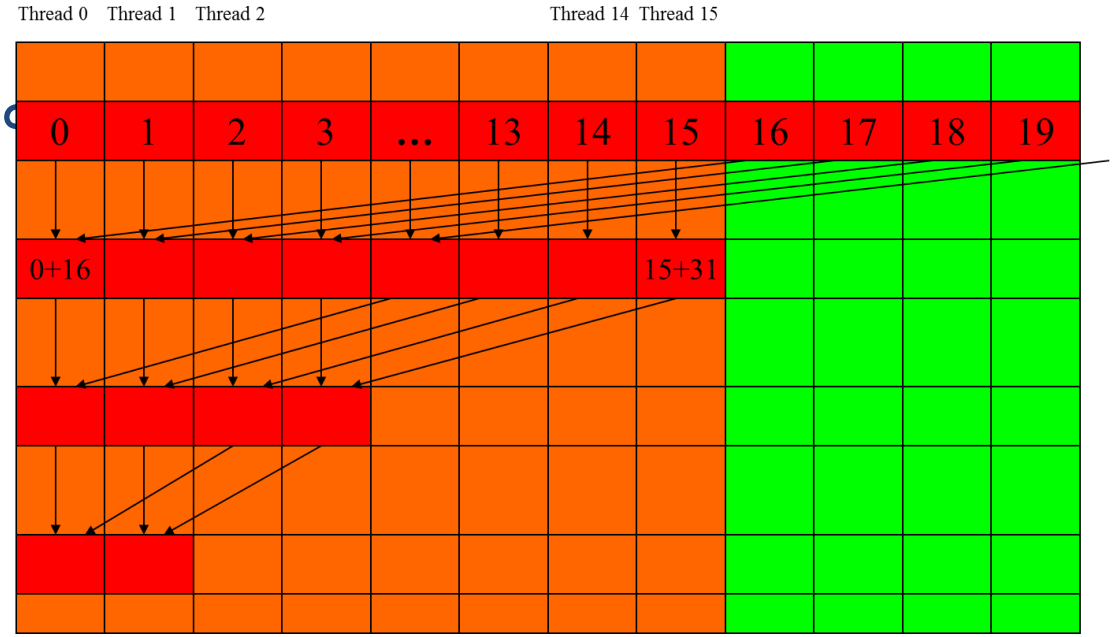
CUDA Parallel Reduction (cont.)

```
__global__
void reduce(float* input, int length, float* output) {

  __shared__ float partialSum[2*BLOCK_SIZE];
  unsigned int t = threadIdx.x;
  unsigned int start = 2*blockIdx.x*blockDim.x;

  partialSum[t] = input[start + t];
  partialSum[blockDim.x+t] = input[start+blockDim.x+t];
```



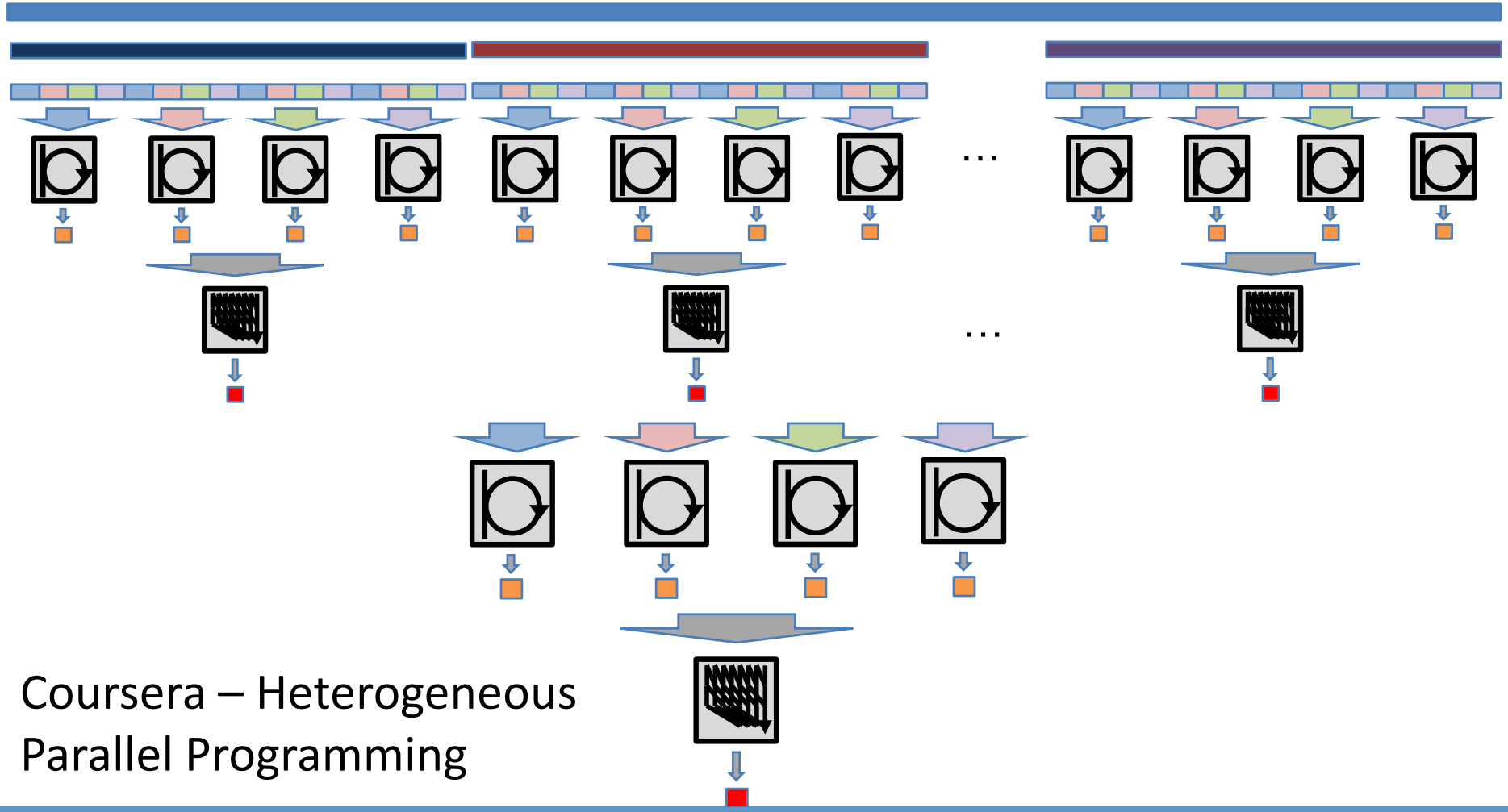
CUDA Parallel Reduction

```
__global
void reduc
```



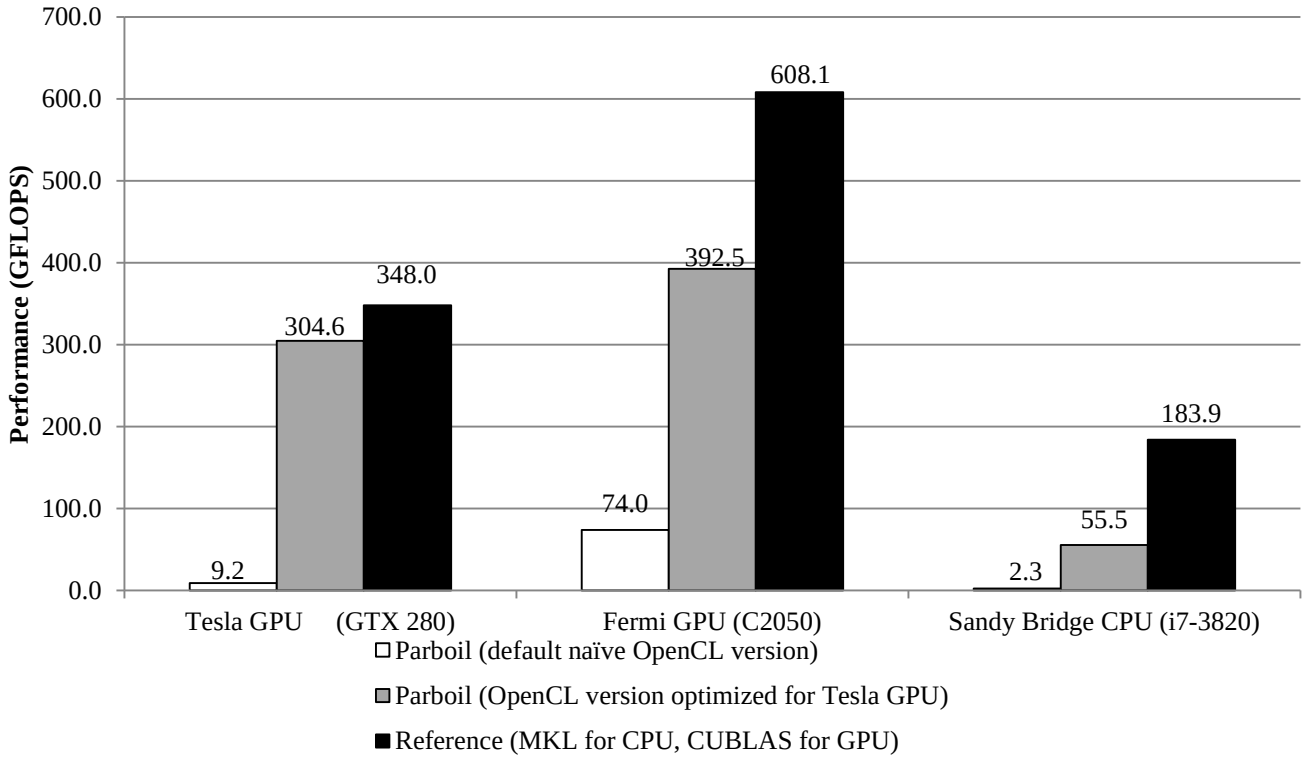

```
for (unsigned int stride = blockDim.x;
    stride > 0;  stride /= 2)
{
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t+stride];
}
```

High-Performance GPU Reduction

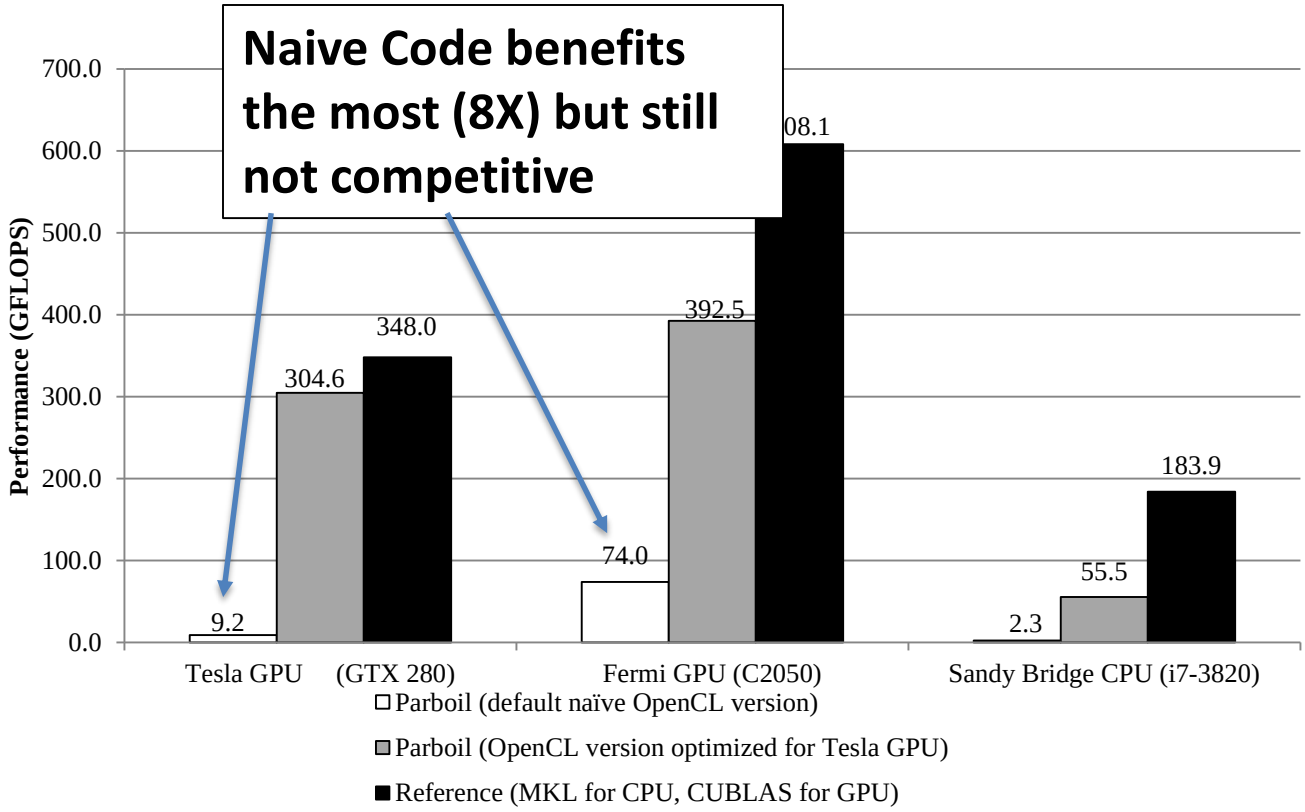


Coursera – Heterogeneous Parallel Programming

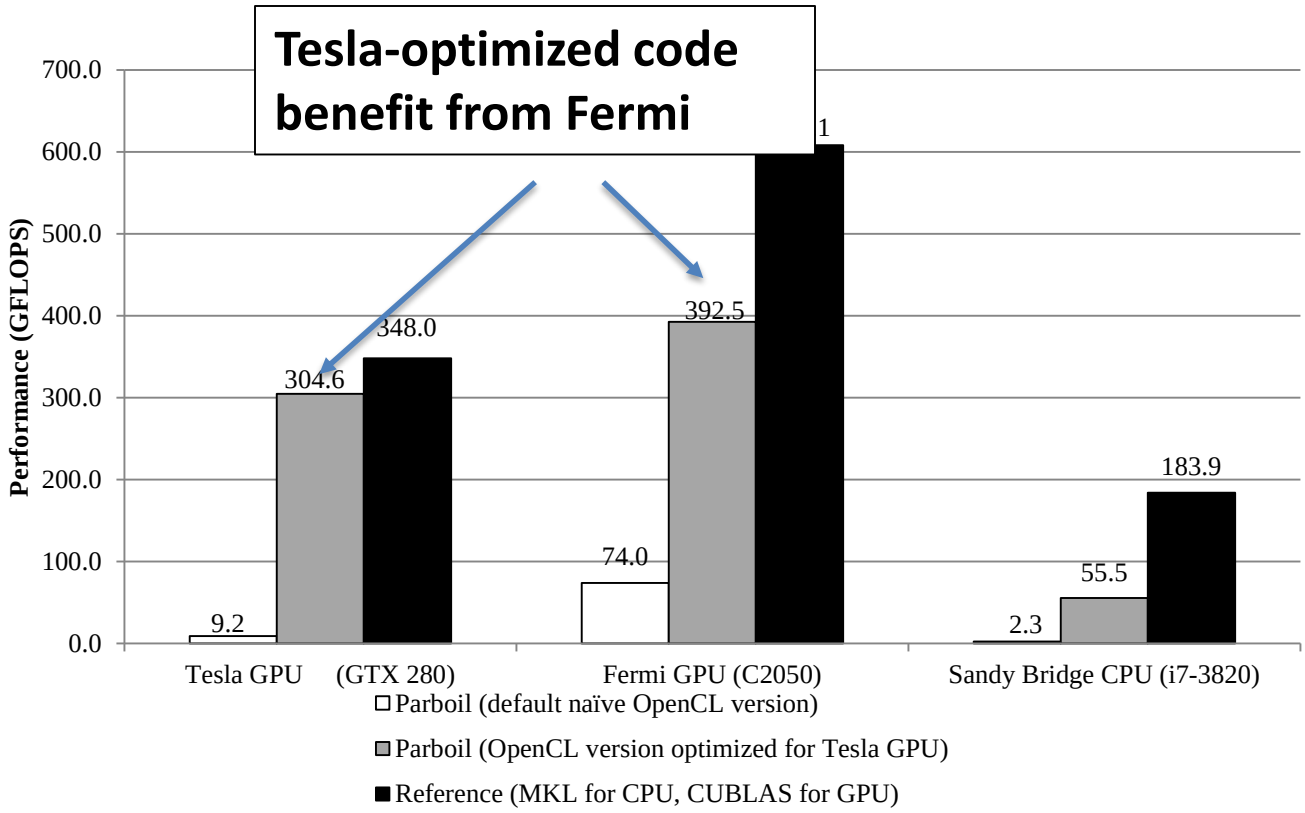
Current State of Performance Portability - DGEMM Case Study



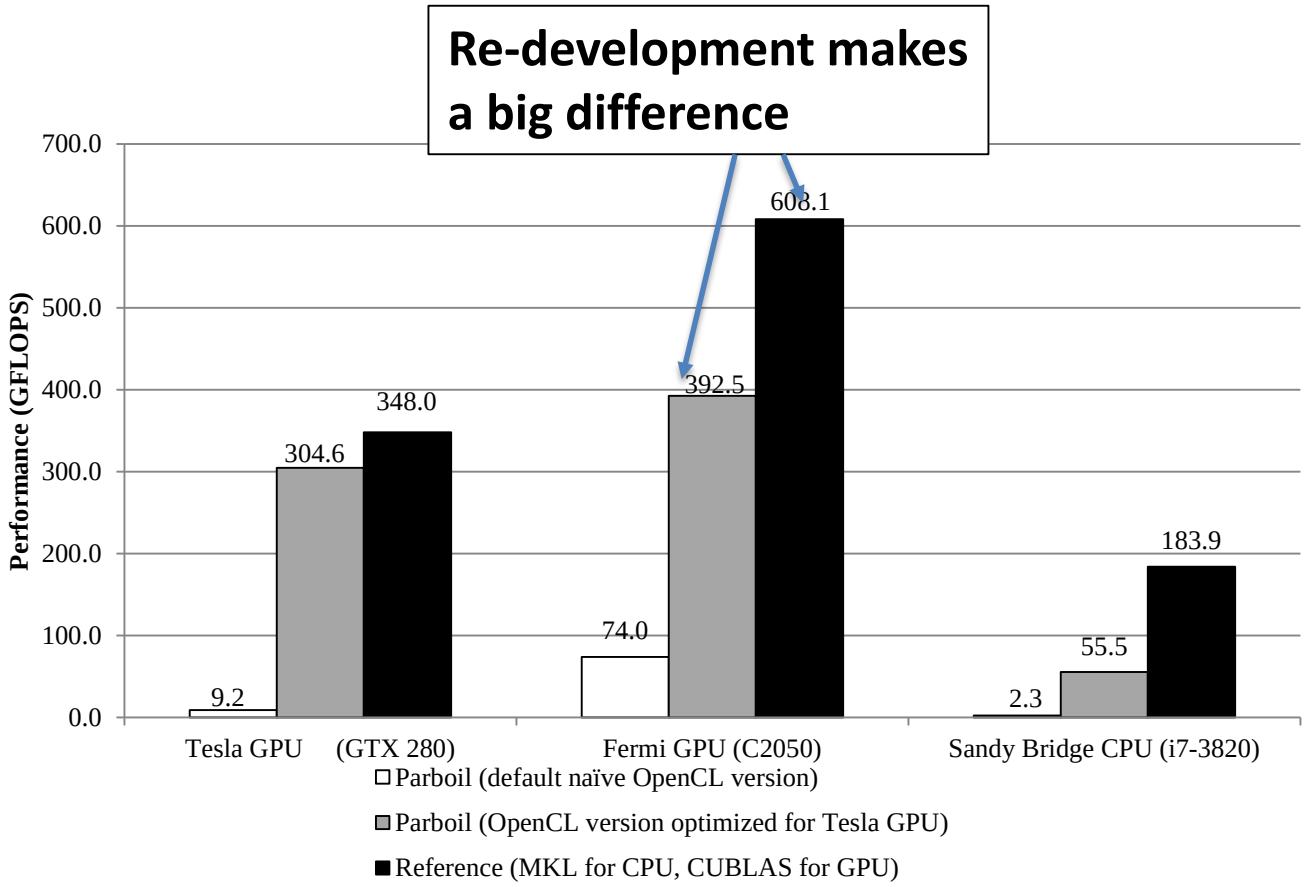
Current State of Performance Portability - DGEMM Case Study



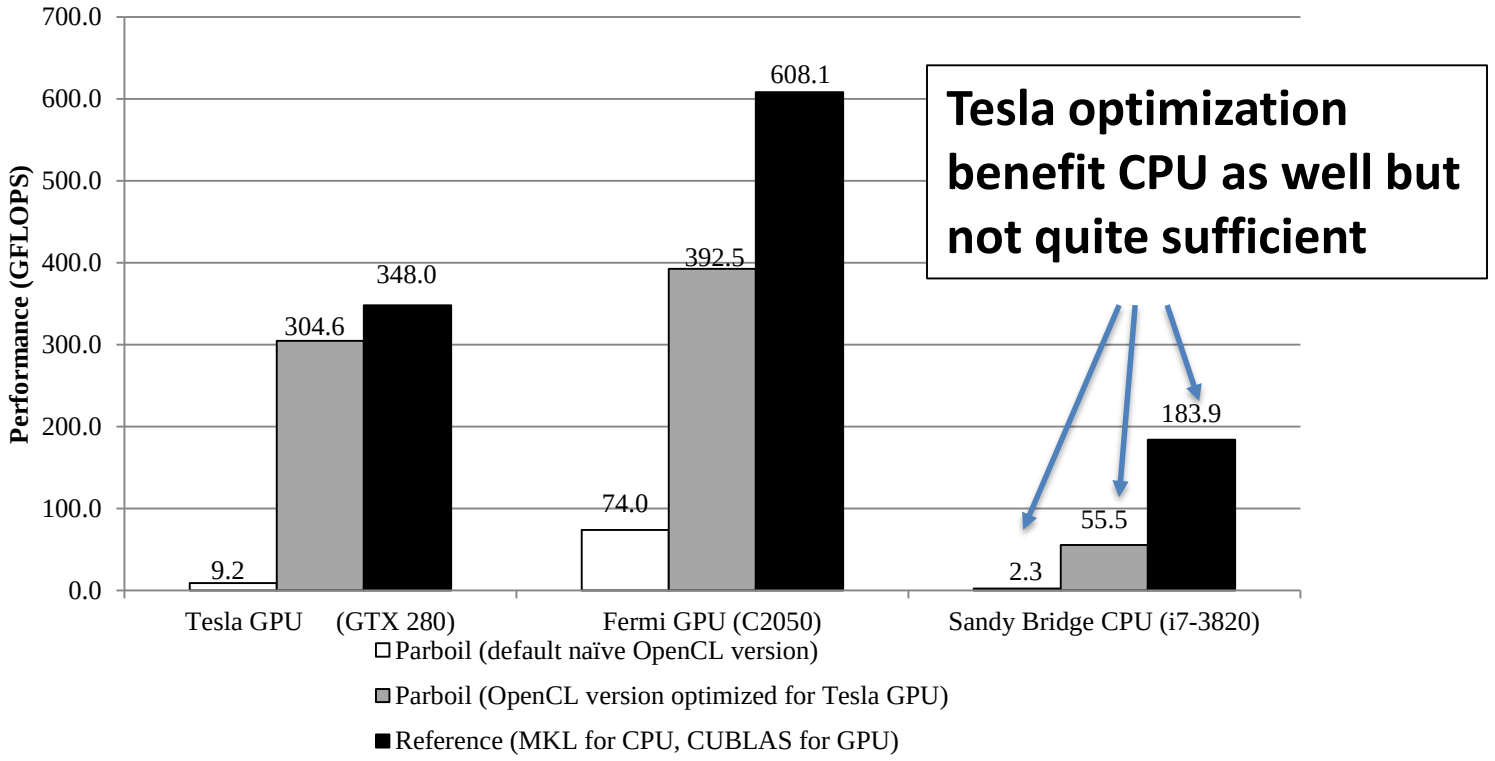
Current State of Performance Portability - DGEMM Case Study



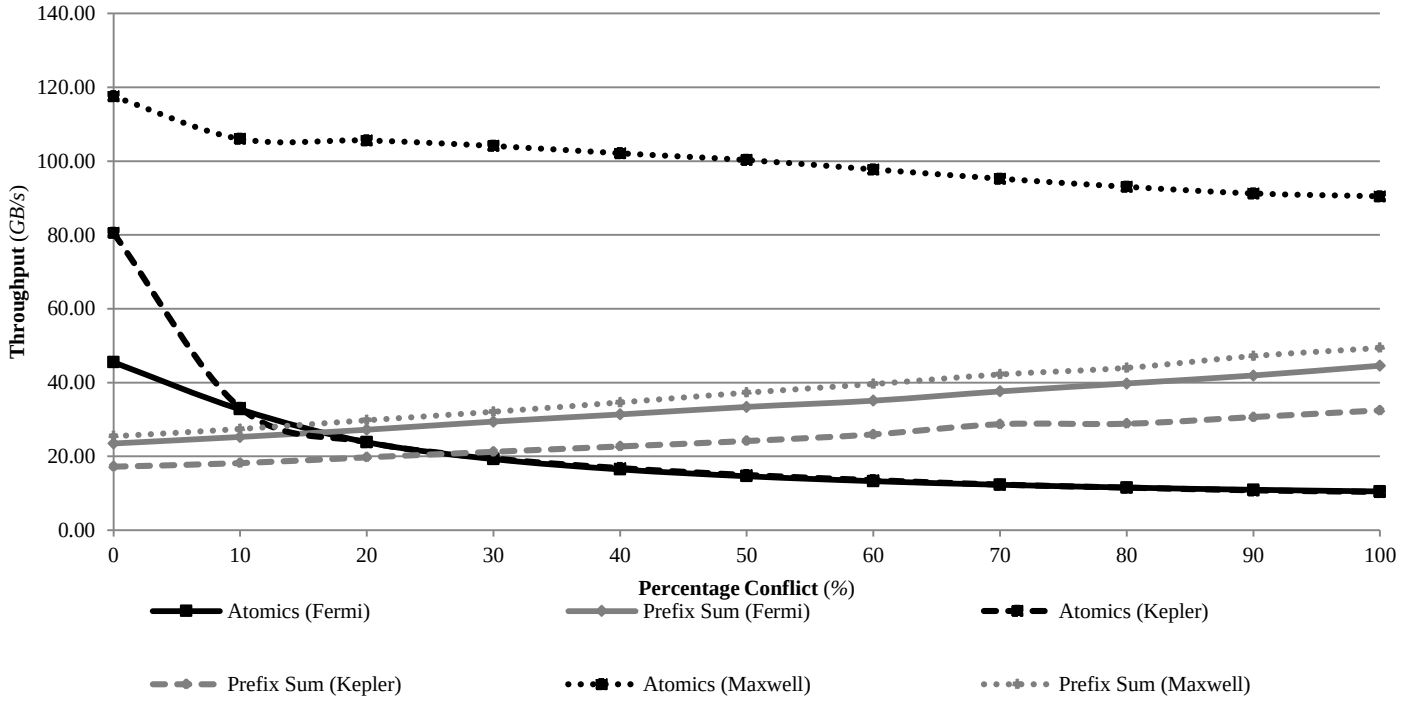
Current State of Performance Portability - DGEMM Case Study



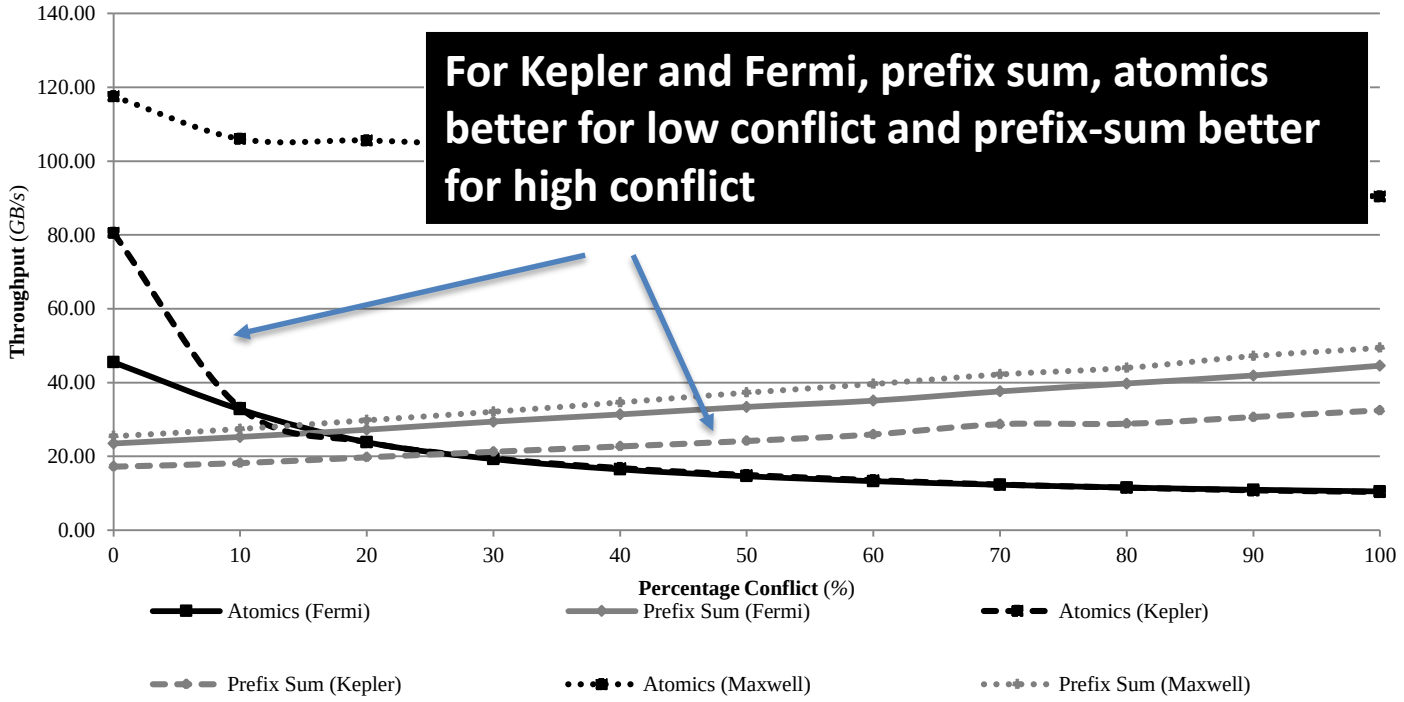
Current State of Performance Portability - DGEMM Case Study



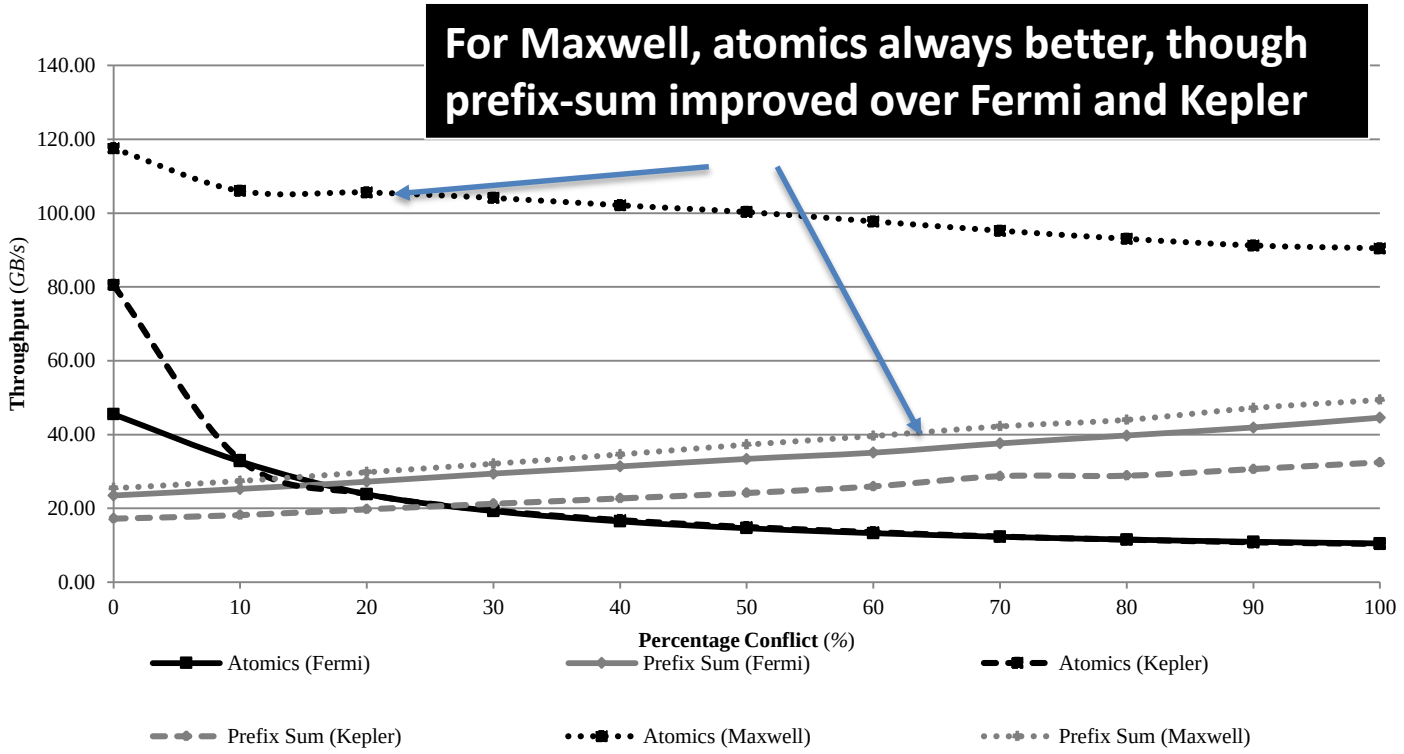
Algorithm Selection
Stream Compaction Case Study



Algorithm Selection
Stream Compaction Case Study



Algorithm Selection
Stream Compaction Case Study



A Practical Programming System for Heterogeneous Platforms

Triolet (Dakkak/El Hajj/Rodrigues)

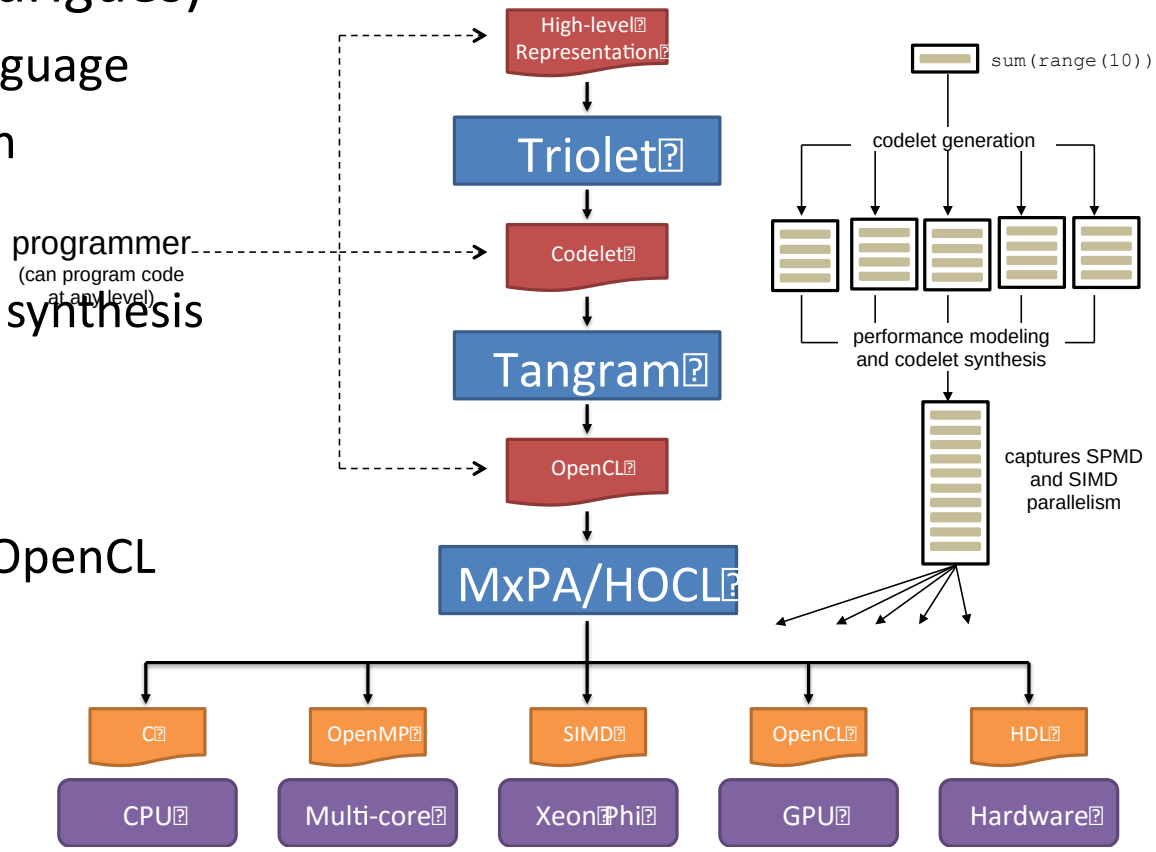
- High-level library-driven language
- Automated data distribution

Tangram (Chang)

- Performance portable code synthesis
- Algorithm-level auto-tuning

MxPA/HOCL (Garcia/Kim)

- Locality-centric scheduling OpenCL compiler
- Dynamic vectorization
- Joint CPU-GPU execution



Tangram

- A language, compiler and runtime
- A C++ extension to support
  - recursive decomposition and over decomposition
  - data placement
    - Using containers, data placement is performed by compiler
  - parameterization
    - Using __tunable keywords
  - pattern replacement
    - Alternative codelets

Tangram Code Example: Reduction

```
__codelet
int reduce(const Array<1,int> in) {
  int len = in.size();
  int accum = 0;
  for(int i=0; i<len; i++) {
    accum += in[i];
  }
  return accum;
}
```

(a) Atomic scalar codelet

```
__codelet __vector __tag(kog)
int reduce(const Array<1,int> in) {
  __shared __tunable Vector<1,int> vec();
  __shared int tmp[vec.size()];
  int len = in.size();
  int id = vec.id();
  tmp[id] = id < len ? in[id] : 0;
  int idle_len = 1;
  while(id >= idle_len) {
    tmp[id] += tmp[id-idle_len];
    idle_len *= 2;
  }
  if(id==0)
    return tmp[vec.size()-1];
}
```

(b) Atomic vector codelet

```
__codelet  __tag(asso_tiled)
int reduce(const Array<1,int> in) {
  __tunable int p;
  int len = in.size();
  int tile_size= (len+p-1)/p;
  return reduce( map( reduce,
                      partition(in,
                                p,
                                sequence(0,tile_size,len),
                                sequence(1),
                                sequence(tile_size, tile_size, len+1))));
}
```

(c) Compound codelet using adjacent tiling

```
__codelet __tag(stride_tiled)
int reduce(const Array<1,int> in) {
  __tunable int p;
  int len = in.size();
  int tile_size= (len+p-1)/p;
  return reduce( map( reduce,
                      partition(in,
                                p,
                                sequence(0,1,p),
                                sequence(p),
                                sequence((p-1)*tile_size, 1, len+1))));
}
```

(d) Compound codelet using strided tiling

Code Example: Reduction

```
__codelet
int reduce(const Array<1,int> in)
{
  int len = in.size();
  int accum = 0;
  for(int i=0; i<len; i++) {
    accum += in[i];
  }
  return accum;
}
```

```
ı+1))));
```

```
    idle_len *= 2;
  }
  if(id==0)
    return tmp[vec.size()-1];
}
```

(a) Atomic scalar codelet

```
partition(in,
          p,
          sequence(0,1,p),
          sequence(p),
          sequence((p-1)*tile_size, 1, len+1))));
}
```

(b) Atomic vector codelet

(d) Compound codelet using strided tiling

Code Example: Reduction

```
__codelet __vector __tag(kog)
int reduce(const Array<1,int> in) {
  __shared __tunable Vector<1,int> vec();
  __shared int tmp[vec.size()];
  int len = in.size();
  int id = vec.id();
  tmp[id] = id < len ? in[id] : 0;
  int idle_len = 1;                                   len+1)));
  while(id >= idle_len) {
    tmp[id] += tmp[id-idle_len];
    idle_len *= 2;
  }
  if(id==0)
    return tmp[vec.size()-1];                         n+1)));
}
```

(b) Atomic vector codelet

Code Example: Reduction

```
__codelet  __tag(asso_tiled)
int reduce(const Array<1,int> in) {
  __tunable int p;
  int len = in.size();
  int tile_size= (len+p-1)/p;
  return reduce( map(reduce,
                     partition(in,
                               p,
                               sequence(0,tile_size,len),
                               sequence(1),
                               sequence(tile_size,tile_size,len+1))));
}
```

(c) Compound codelet using adjacent tiling

Built-in function partition(*c,n,start,inc,end*)
Returns *n* sub-containers *c[i]* of *c* where *c[i]* goes from *start[i]* to *end[i]* with increment *inc[i]*

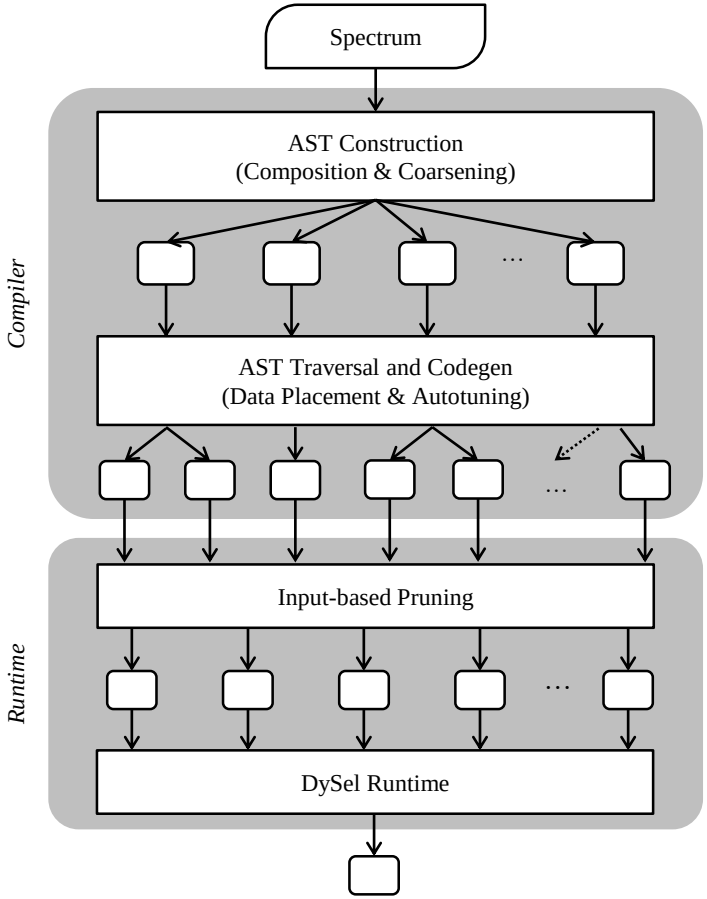
Code Example: Reduction

```
__codelet __tag(stride_tiled)
int reduce(const Array<1,int> in) {
  __tunable int p;
  int len = in.size();
  int tile_size= (len+p-1)/p;
  return reduce( map( reduce,
                      partition(in,
                                p,
                                sequence(0,1,p),
                                sequence(p),
                                sequence((p-1)*tile_size, 1, len+1))));
}
```

(d) Compound codelet using strided tiling

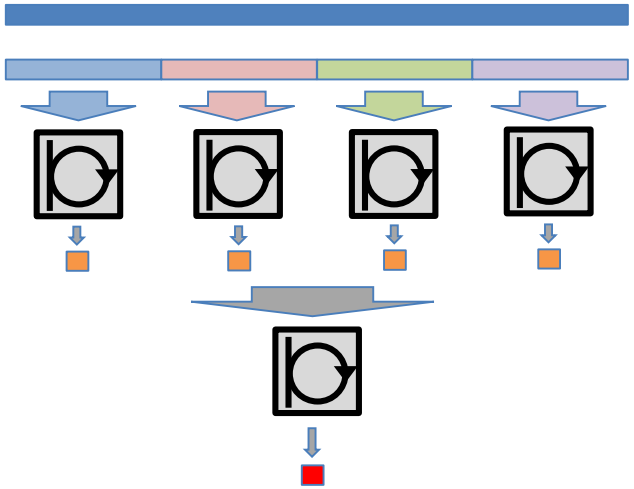
Tangram Workflow

- Construction OpenCL AST from Tangram AST
- Generate few competitive versions for runtime using relative merits (parallelism and locality, for example)
- DySel Runtime applies micro-profiling and dynamically selects best version for the actual data and hardware

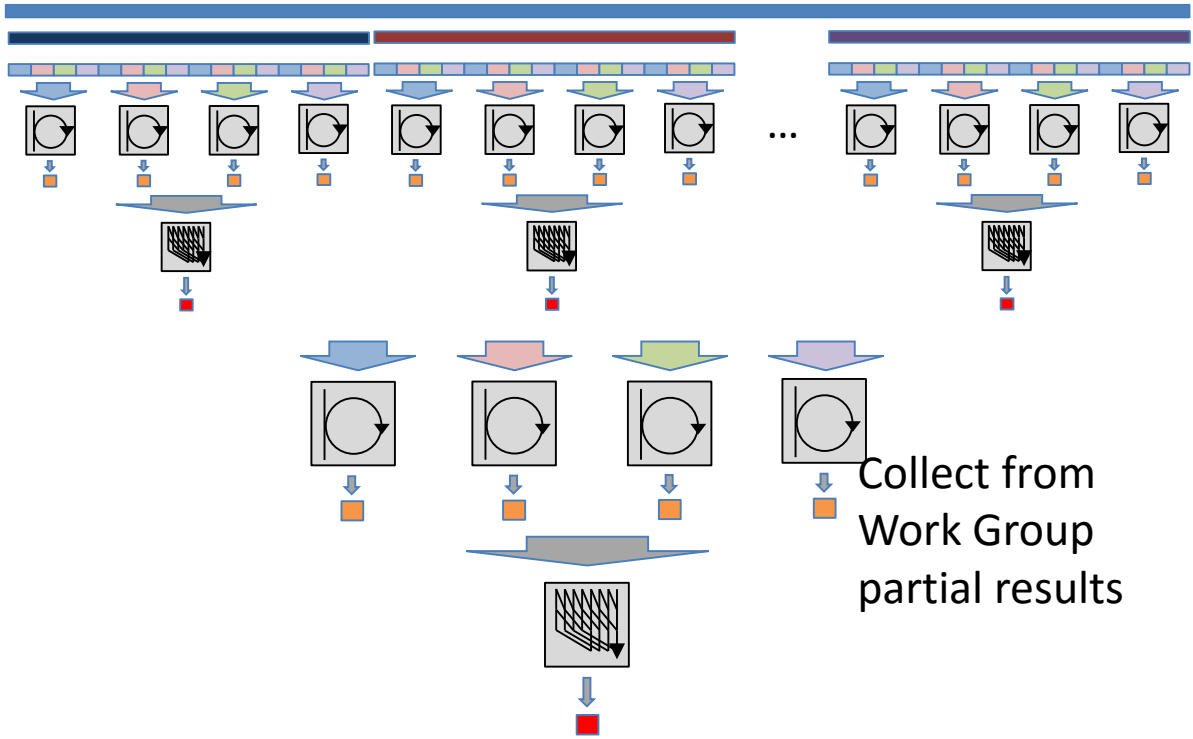


Reduction – CPU vs. GPU (Part 2)

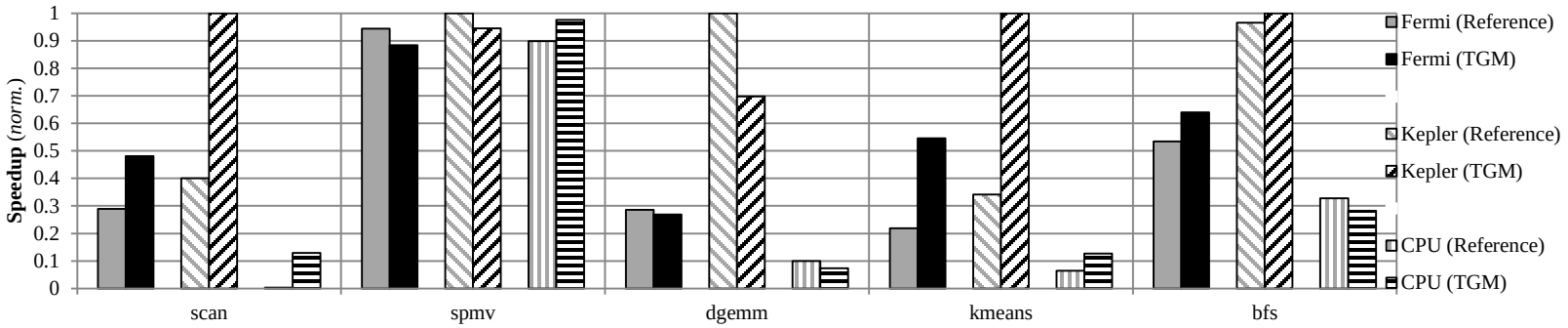
CPU 2-level hierarchy



GPU 4-level hierarchy



Experimental Results



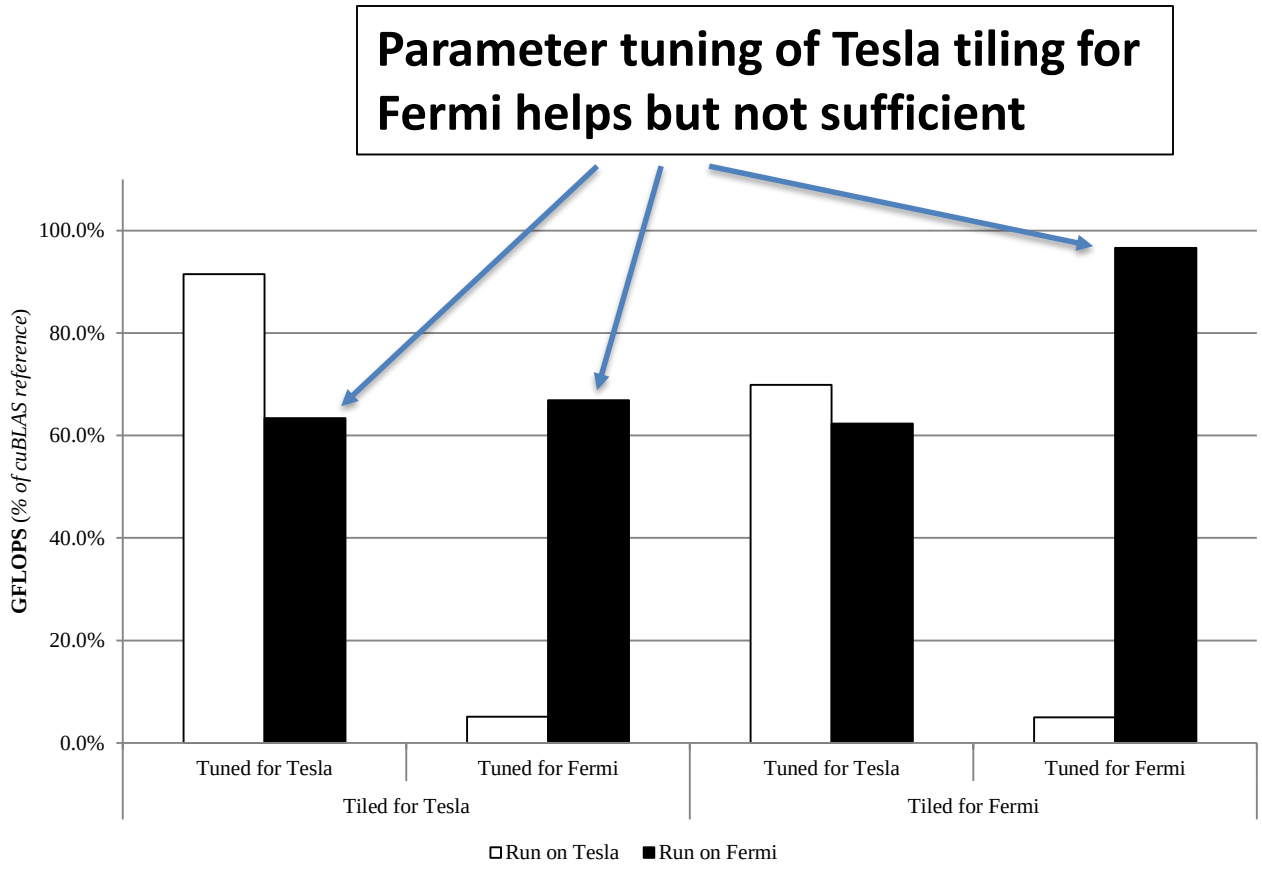
- We achieve at least 70% of reference libraries (MKL, CUBLAS, CUSPARSE, Thrust) and reference benchmark suite (Rodinia)

Summary

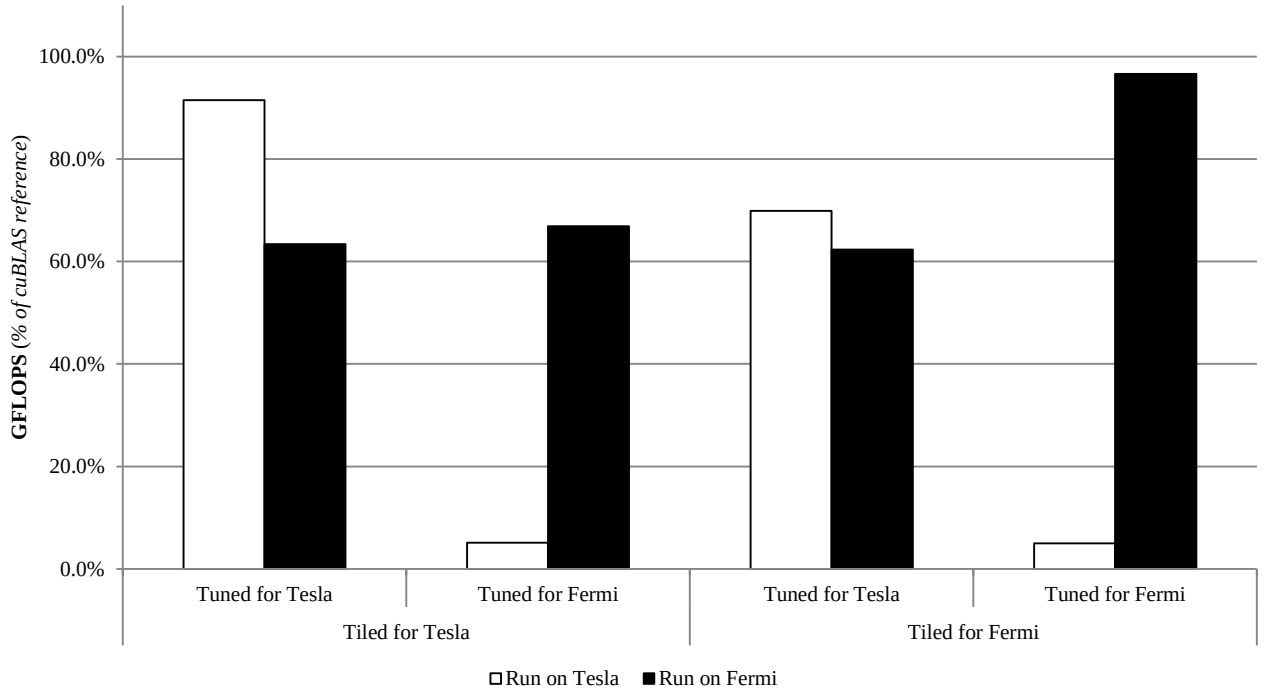
- Heterogeneous computing gaining importance
  - Performance, energy advantages
  - Heterogeneity increasing in both memory and processors
- Programming for heterogeneous computing evolving
  - Currently low-level interfaces – CUDA, OpenCL
  - Next higher-level – OpenACC, Parallel C++
  - Ultimately need code synthesis - Tangram

THANK YOU FOR YOUR ATTENTION!
QUESTIONS?

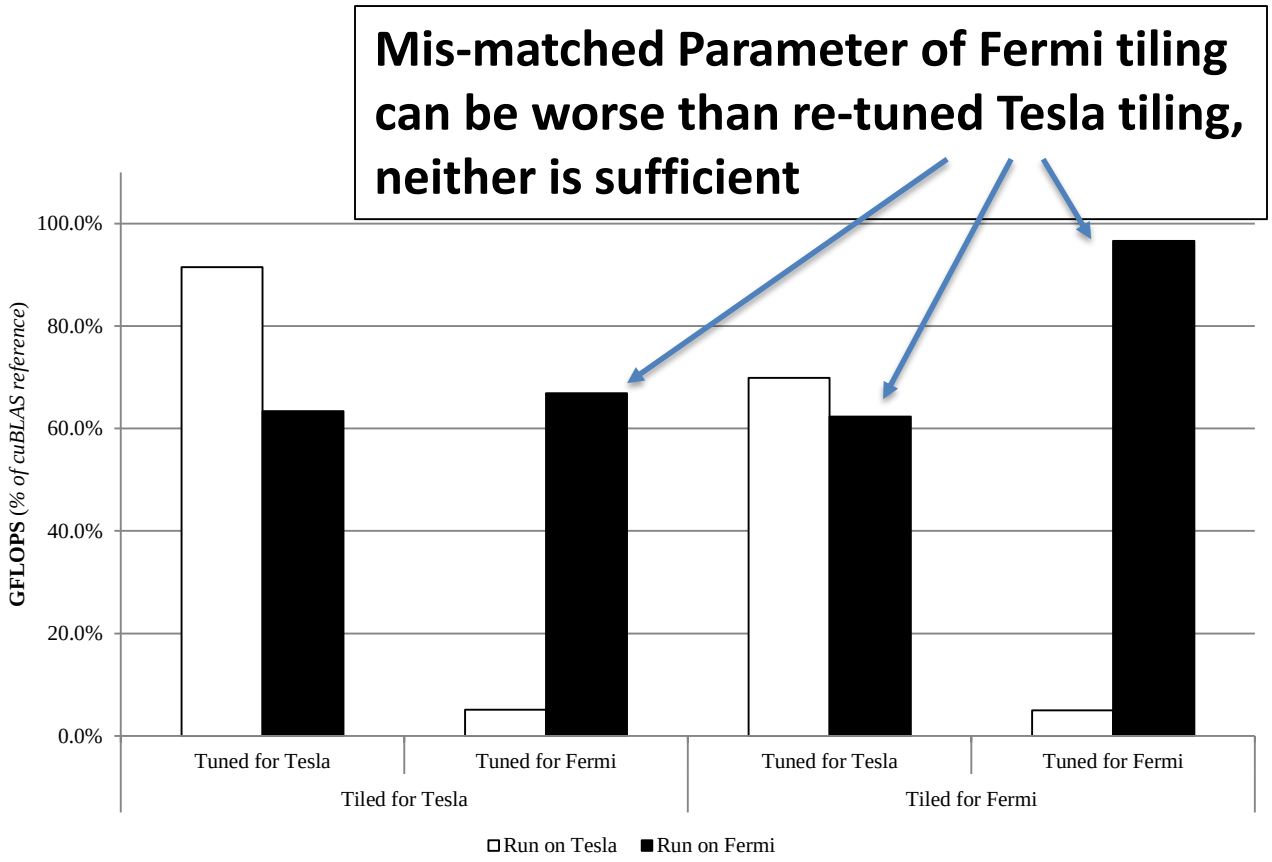
Data Tiling Performance Portability - DGEMM case study



Data Tiling Performance Portability - DGEMM case study



Data Tiling Performance Portability - DGEMM case study



Data Tiling Performance Portability - DGEMM case study

Mis-matched Parameter of Tesla tiling can be worse than re-tuned Fermi tiling, neither is sufficient