Unstructured Computations on Knights Landing Architecture

Mohammed A. Al Farhan and David E. Keyes

Extreme Computing Research Center
King Abdullah University of Science and Technology

Intel Extreme Performance Users Group (IXPUG)
Middle East Conference 2018 at KAUST
April 22-25, 2018 • Thuwal, Saudi Arabia

April 24, 2018

Highlights of the Contributions

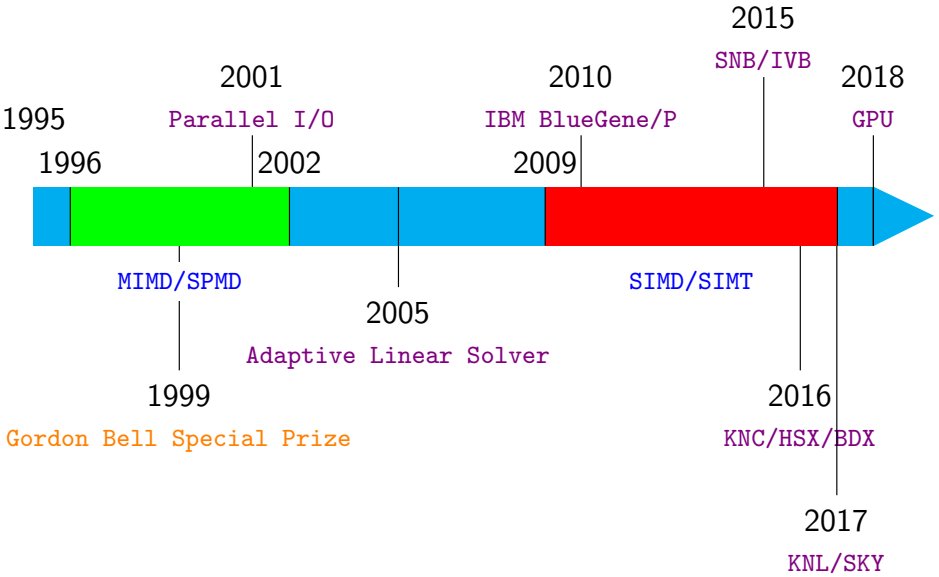
- Address the classical tension between unstructured data (indirect addressing) and highly structured architectures (vector instructions)
- Algorithm: nonlinearly implicit pseudo-transient Newton-Krylov-Schwarz solver
- Application: computational aerodynamics (Gordon Bell winning NASA legacy code)
- Architecture: Intel Xeon Phi "Knights Landing"
- AVX-512CD instructions provide new solutions for conflicting objectives
- Independence of writes versus memory locality
- Finest-grain implementation of PETSc-FUN3D to exploit language features
- Gains in wall-clock time 2.9x over scalar compilation for strong thread scaling
- Distributed-memory not addressed here; proven for two decades in weak scaling
- Gains in energy-to-solution for a given wall-clock time up to 1.7x

Outline

Fully Unstructured Navier-Stokes in 3D

- An unstructured tetrahedral mesh Euler and Navier-Stokes research code, which is closely related to the export-controlled state-of-the-practice from NASA Langley Research Center
- For over two decades FUN3D has been under active development for modeling fluid flow, and design optimization of airplanes, automobiles, and submarines with a number of vertices up to 10 billion

PETSc-FUN3D


1995
1996
2001
Parallel I/O
2002
2005
Adaptive Linear Solver
2009
2010
IBM BlueGene/P
2015
SNB/IVB
2016
KNC/HSX/BDX
2017
KNL/SKY
2018
GPU
MIMD/SPMD
1999
Gordon Bell Special Prize
SIMD/SIMT

Edge-based Loop Kernel – Sequential

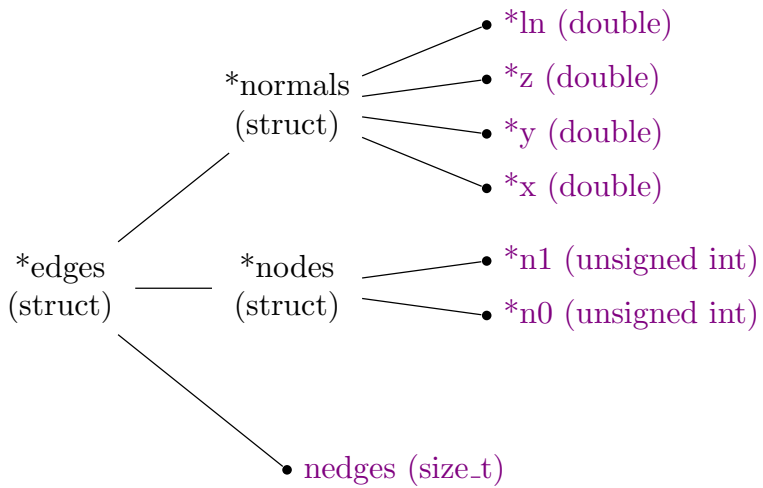
1: **for all** $e \in$ array of edges **do**
2: Get the index of the right node of e
3: Get the index of the left node of e
4: Read the flow variables from both endpoints of e
5: Compute the flux and the residual
6: Perform "write-back" operations to update the flow variables of e
7: **end for**

Spatial and Temporal Locality of Reference

- Left endpoints of the edges are ordered in ascending order
  - The right endpoints and the edges' components are ordered accordingly
- Traversing the edges index table is done through one iterator pointer that is sequentially incremented
- A kernel loop that iterates over the stencil data items is essentially transformed from a loop over the edges into a loop over the vertices, in which the iterative traversing is linearly based upon the left endpoints

Geometric Data Layout

- Tree of Struct-of-Arrays (SoA)



Residual and Gradient Data Layout

- FROM Array-of-Structs-of-Arrays (AoSoA) TO Array-of-Structs-of-Strided-Arrays

(A)

struct {

X:	u	v	w	p	.....	u	v	w	p
Y:	u	v	w	p	.....	u	v	w	p
Z:	u	v	w	p	.....	u	v	w	p

} Gradient;

(B)

struct {

Q:	u	v	w	p	.....	u	v	w	p

} Residual;

- Alignment and padding to 64-bytes cache line boundaries
- Contiguously adjust every 4 DoF next to each other (u, v, w, p)

Memory-aware Allocation

- Explicit heap allocator based on how frequent the data are being accessed
  - Intel *memkind* heap manager library and *jemalloc* library
    - `hbw_posix_memalign_psize()`

Thread-level Parallelism

- Edges workloads is partitioned using multilevel k-way partitioning scheme of METIS
- METIS searches for an adequate load balance across the OpenMP threads
- It minimizes the aggregate cross edges' weight (i.e., reduce the edge-cut)
- Reverse Cuthill-McKee (RCM) is used for reordering the vertices
- Work replication – METIS distribution conserves thread safety without the need of atomic
- Only the "master" thread is allowed to perform the write-back operation – "Owner Compute"
- The METIS strategy of redundancy in computations purges the overheads of using global synchronizing barrier between the working threads

Edge-based Loop – Optimized and Threaded

```
1 #pragma omp parallel
2 {
3   const uint32_t t = omp_get_thread_num();
4   for(uint32_t i = ie[t]; i < ie[t+1]; i++){
5     /*Load and compute*/
6     if(parts[n0[i]] == t){/*Write-back into v[n0[i]]*/}
7     if(parts[n1[i]] == t){/*Write-back into v[n1[i]]*/}
8   }
9 }
```

Intel AVX-512 Instruction Set Architecture

- Knights Landing
- Skylake

- AVX512F: Foundation
- AVX512CD: Conflict Detection

Intel AVX-512 Instruction Set Architecture

- Knights Landing
- Skylake

- AVX512ER: Exponential and Reciprocal
- AVX512PF: Prefetching

Intel AVX-512 Instruction Set Architecture

- Knights Landing
- Skylake

- AVX512VL: Vector Length
- AVX512DQ: Doubleword and Quadword
- AVX512BW: Byte and Word

Vectorizing Edge-based Loop Kernels

- Read: Load and gather instructions
- Arithmetic: FMA (`vfmadd/vfmsub/vfnmadd`)
- Control-flow to Data-flow: Masking instructions
- Square Root/Division: Reciprocal instructions
  - $\sqrt{x}$: `_mm512_mul_pd(_mm512_rsqrtXX_pd(x),x)`
  - $\frac{c}{x}$: `_mm512_mul_pd(_mm512_rcpXX_pd(x),c)`
- Prefetching: Software gather prefetching instructions
- Write-back: Conflict detection and scatter instructions

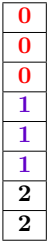
Conflict Detection Instructions

Conflict-free Array

Array with Conflicts

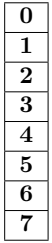
Conflict Detection Instructions

Conflict-free Array

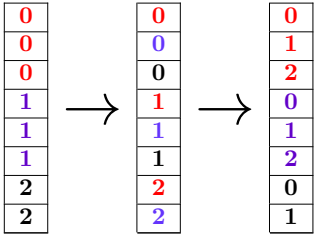


Array with Conflicts

Conflict Detection Instructions



Conflict-free Array



Array with Conflicts

Conflict Detection Instructions

1. Generate an initial mask based on the thread ID and node index
2. Scan the SIMD lane to identify the conflicted data
3. Generate a mask that separates a conflict-free data subset
4. Perform a safe gather mask operation (load) to generate registers with contiguous data items
5. Update the operands with multiple FMA instructions (number of double precision arithmetics)
6. Perform a safe scatter mask operation (write-back) to sprinkle the updated data items over the memory addresses based on their indices
7. Perform SIMD boolean operation to "mask out" the already written indices from the mask (swizzling)
8. Repeat the aforementioned steps again on all of the remaining subsets within the SIMD lane

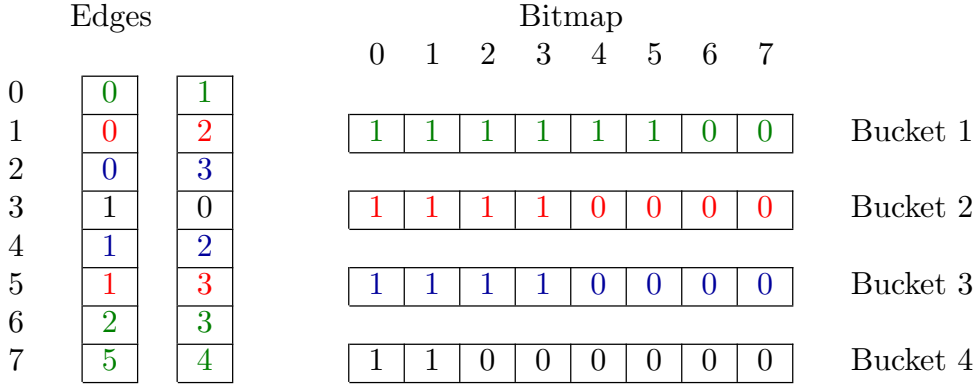
Vectorized Edge-based Loop Kernel

```
1  #pragma omp parallel
2  {
3    const uint32_t t = omp_get_thread_num();
4    const uint32_t l = ie[t+1] - ((ie[t+1]-ie[t]) % 8);
5    for(uint32_t i = ie[t]; i < l; i += 8){
6      /*Load and compute on the SIMD lane elements*/
7
8      /*The Write-back inner loop*/
9      _mm512_cmpeq_epi32_mask(/*...*/);
10     do {
11       _mm512_mask_conflict_epi32(/*...*/);
12       _mm512_broadcastmw_epi32(/*...*/);
13       _mm512_mask_testn_epi32_mask(/*...*/);
14       /*Gather, compute, and scatter*/
15       _mm512_kxor(/*...*/);
16     } while(/*...*/);
17   }
18   /*Peel and remainder loop*/
19   for(uint32_t i = l; i < ie[t+1]; i++){
20     /*load and compute*/
21     if(parts[n0[i]] == t){/*Write-back into v[n0[i]]*/}
22     if(parts[n1[i]] == t){/*Write-back into v[n1[i]]*/}
23   }
24 }
```

Fine-grained Data Partitioning

- Improve the vectorization efficiency and increase the size of the independent data subset within a SIMD lane
- Bucket sort routine based on a variant of the edge coloring algorithm
- Aim to:
  - Extract independent subsets of the edges
  - Prune the overhead of the innermost CD loop
- Extract at most 8 independent edges within a bucket
  - To avoid a potential disruption of the initial ordering for cache locality efficiency

Bucket Sort Based on Partial Coloring



Bucket Sort Based on Partial Coloring

1: Create an `nedges_per_color` array to track the colors
2: Create a bitmap to track the endpoints coloring scheme
3: **for** $c \leftarrow 0, k \leftarrow 0, i \leftarrow 0$ to $nndges$ **do**
4: Clear all of the `bitmap` bits
5: **for** $j \leftarrow i$ to $nndges$ **do**
6: Read $n0[j], n1[j]$ bits from the `bitmap` $\rightarrow b0, b1$
7: **if** $b0 \vee b1$ is TRUE **then**
8: CONTINUE
9: **end if**
10: **if** `nedges_per_color[c]` $= 8$ **then**
11: BREAK
12: **end if**
13: Set $n0[j], n1[j]$ bits in bitmap
14: Swap edge data from index j to k and vice versa
15: Increment k and `nedges_per_color[c]` by 1
16: **end for**
17: Increment c by 1
18: $i \leftarrow k$
19: **end for**

Mesh size[1]

- Vertices: 2,761,774
- DoFs: 11,047,096
- Edges: 18,945,809

[1] In 1999 Gordon Bell prize paper, this was considered to be a large mesh but it is hosted conveniently today within a single node

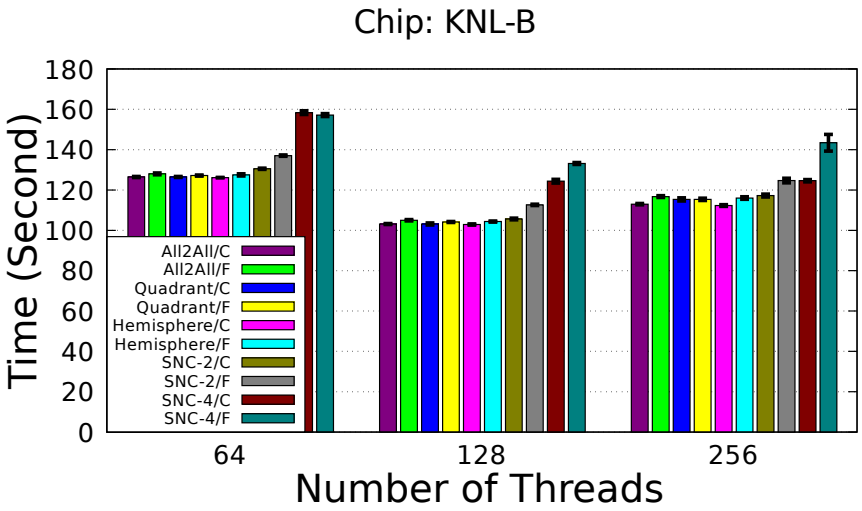
Experimental Design

- Sample space: 20 independent experiments
- Runtime:
  - API: RDSTC instructions and POSIX timestamps
  - Results summary: arithmetic mean
- Memory bandwidth and FLOPS:
  - Bandwidth: MC/EDC RPQ and WPQ inserts
  - Flops: count the arithmetic operations manually
  - Results summary: harmonic mean
- For every experiment, the source code is recompiled, and the memory and cache are completely flushed
- +/- standard deviation of the mean for each experimental sample

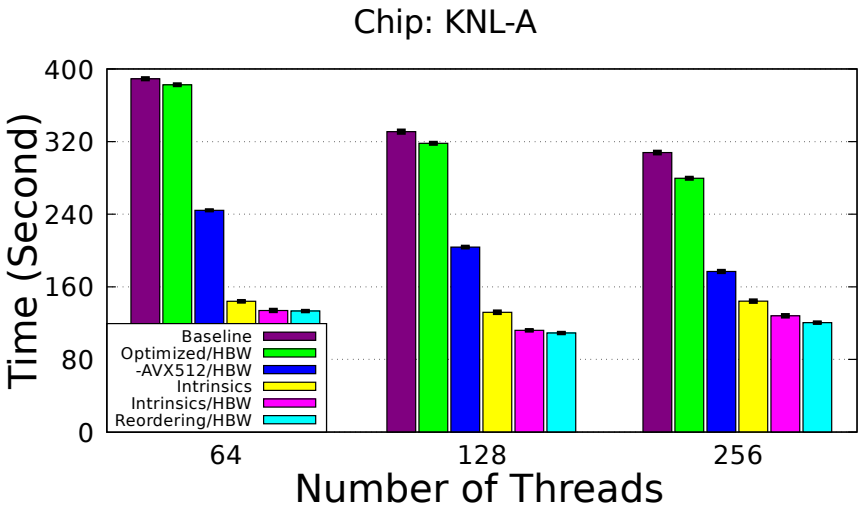
Hardware Specifications

	KNL-A	KNL-B	KNL-C	HSX	BDX	SKY
Family	x200	x200	x200	E5V3	E5V4	Scalable
Model	7210	7210	7290	2699	2680	8176
Socket(s)	1	1	1	2	2	2
Cores	64	64	72	36	28	56
GHz	1.30	1.30	1.50	2.30	2.40	2.10
Watts/socket	215	215	245	145	120	165
DDR4 (GB)	96	96	192	264	132	264
Freq. Driver*	I	A	A	A	I	A
Max GHz	1.50	1.30	1.50	2.30	3.30	2.10
Governor**	P	C	C	O	P	O
Turbo Boost	×	✓	✓	✓	✓	✓

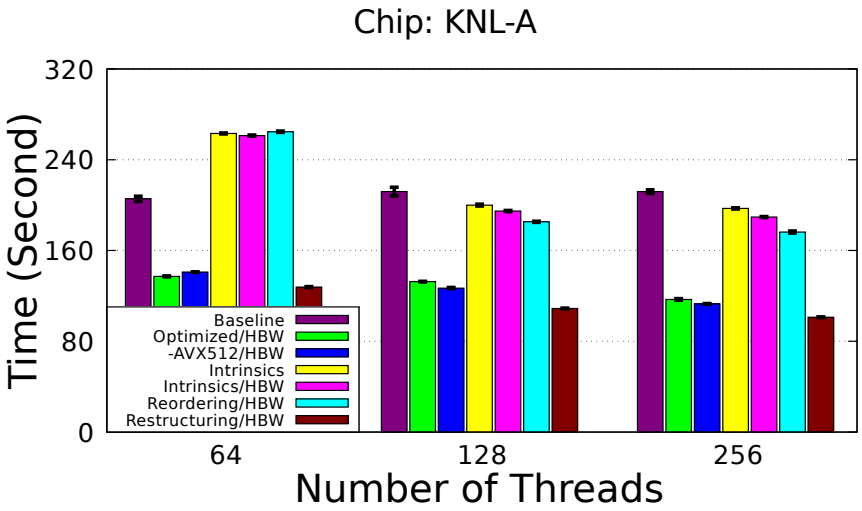
Flux Performance with Different KNL Modes



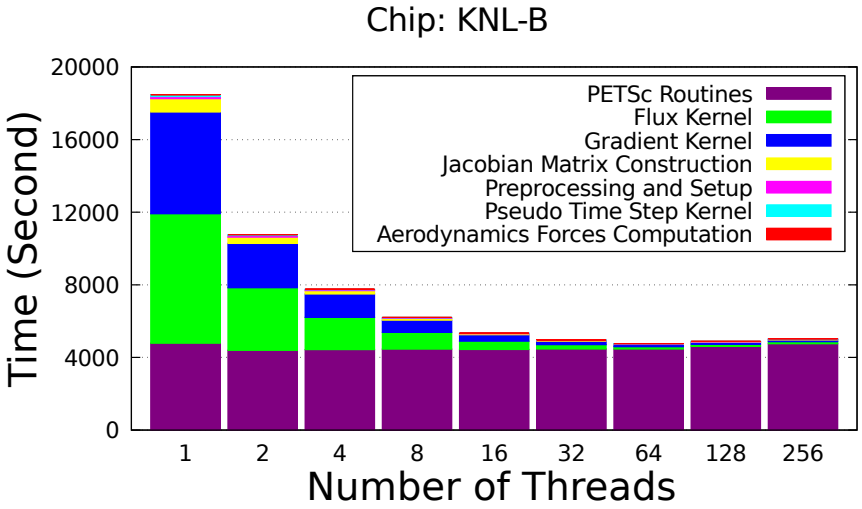
Optimizations of the Flux Kernel



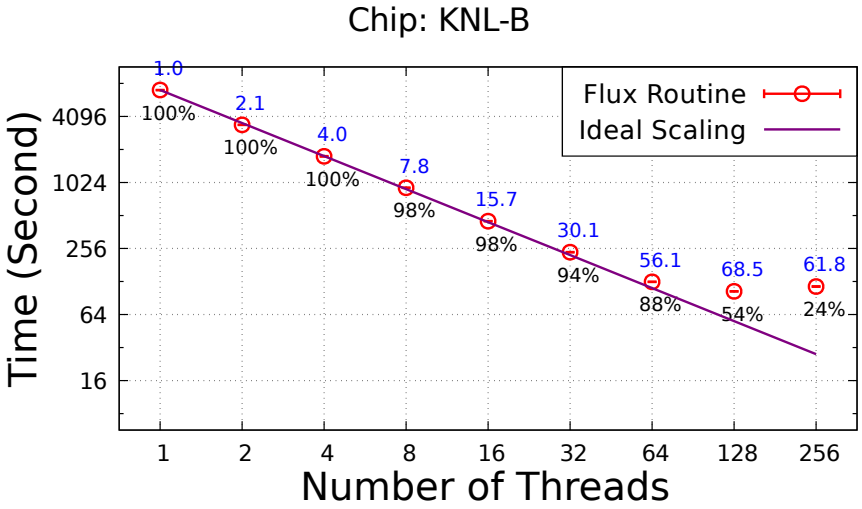
Optimizations of the Gradient Kernel



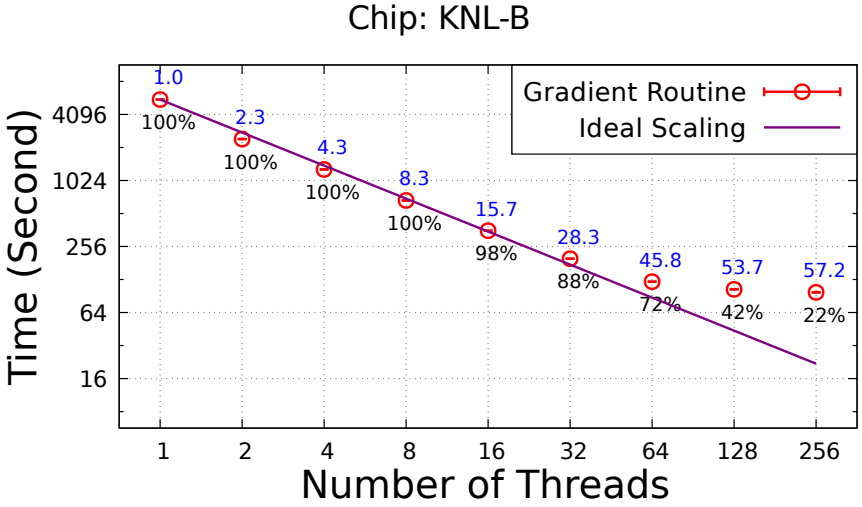
Strong Thread Scalability



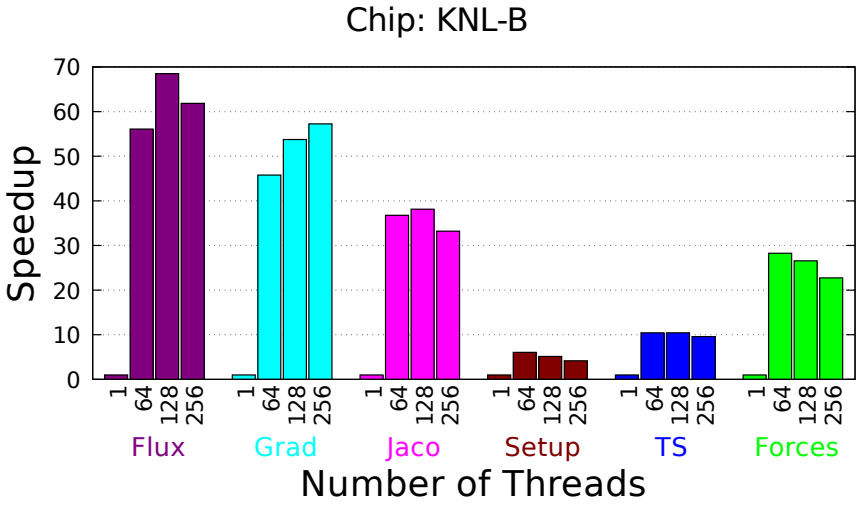
Strong Thread Scalability (Flux Kernel)



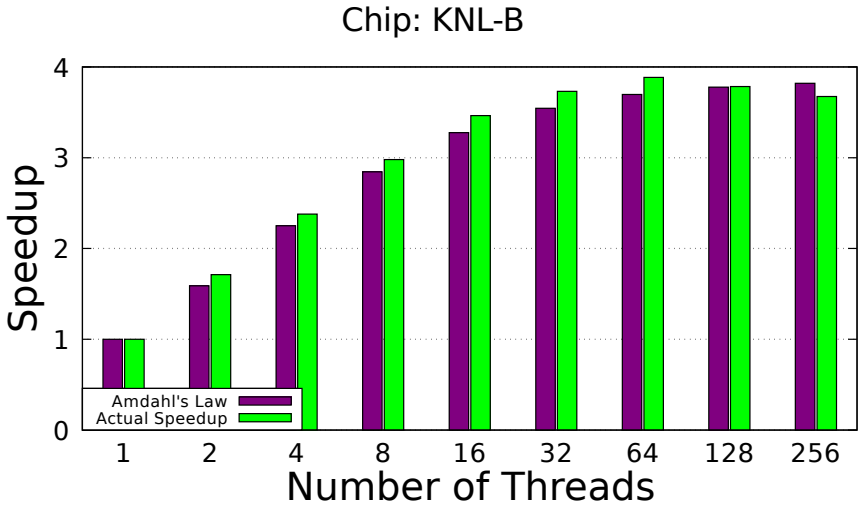
Strong Thread Scalability (Gradient Kernel)



Performance Speedup



Amdahl's Law


Chip: KNL-B
Speedup
Number of Threads
0
1
2
3
4
1
2
4
8
16
32
64
128
256
Amdahl's Law
Actual Speedup

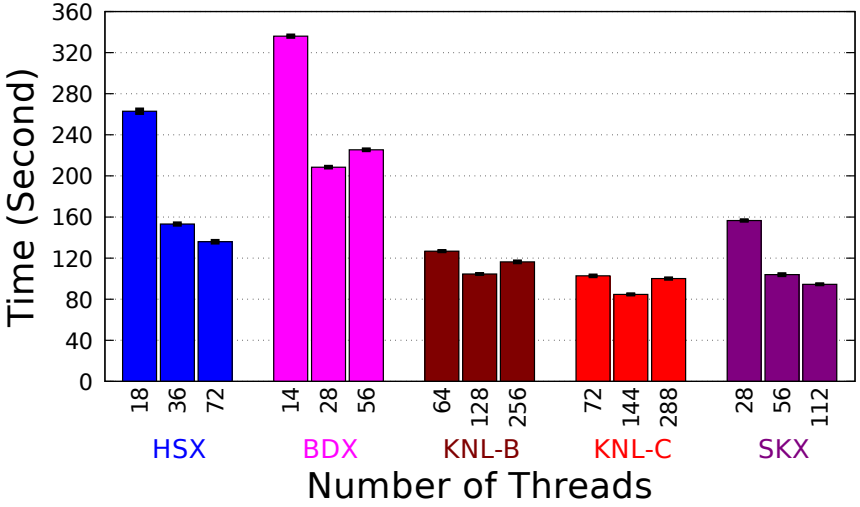
Memory Bandwidth and Flops Performance

	Flux Kernel			Gradient Kernel		
Threads	72	144	288	72	144	288
GFlop/s	117	144	123	21	24	25

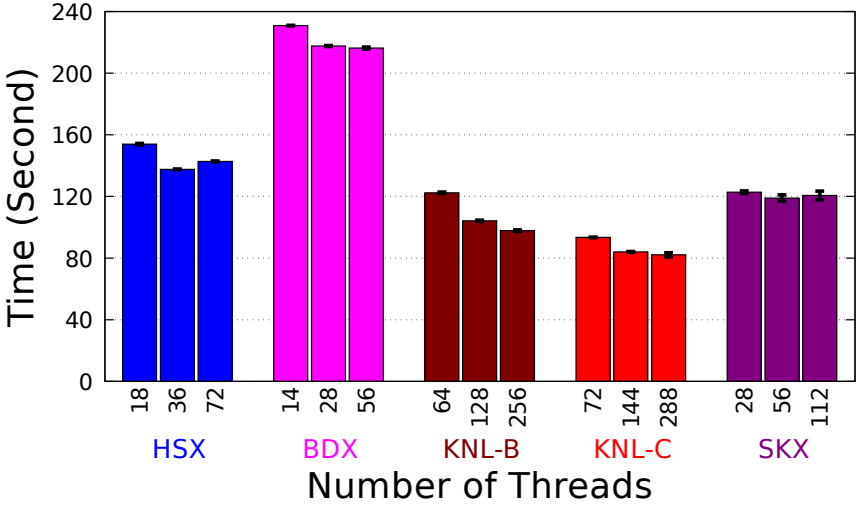
		Flux Kernel			Gradient Kernel		
Threads		72	144	288	72	144	288
DDR	Read: GB/s	14	17	17	8	10	12
	Write: GB/s	7	9	9	0.2	0.4	1
HBW	Read: GB/s	40	51	45	43	53	54
	Write: GB/s	0.1	0.1	0.1	18	22	25

- GFlop/s: 5% out of 3 TFlop/s
- GB/s:
  - DRAM: 25% out of STREAM (77 GB/s (R), 36 GB/s (W))
  - MCDRAM: 17% out of STREAM (314 GB/s (R), 171 GB/s (W))

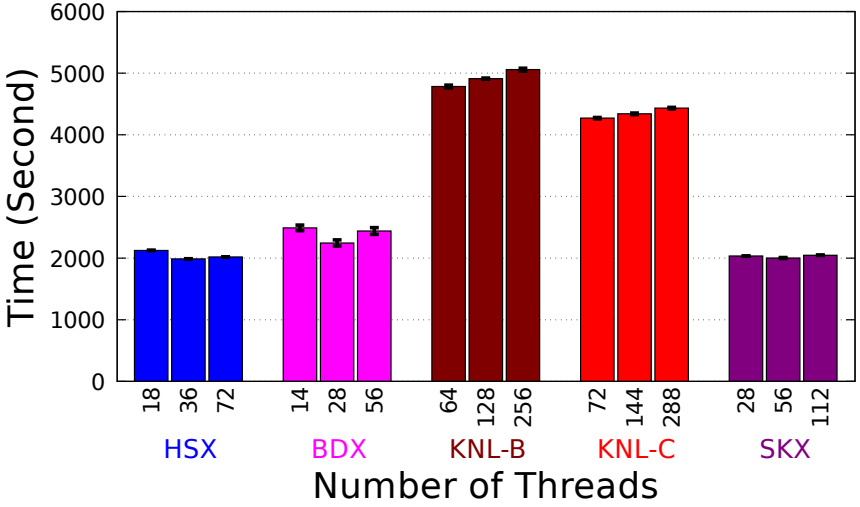
Flux Performance on x86 Hardware



Gradient Performance on x86 Hardware



PETSc-FUN3D Performance on x86 Hardware



Conclusion

- Demonstrate several shared-memory optimizations to extract:
  1. **Thread-level parallelism** – Careful workload distributions and load balancing
  2. **Data-level parallelism** – Utilizing the capabilities of AVX-512 ISA
- Achieve 2.9x speedup in the performance of the flux kernel relative to the baseline code
- Exhibit almost linear scalability up to the full core count of KNL (64 cores), and continued scalability with SMT
- Maintain on Skylake roughly similar speedup with less power consumption [KNL: 245 Watts; SKX: 330 Watts]

Conclusion

- Demonstrate several shared-memory optimizations to extract:
  1. **Thread-level parallelism** – Careful workload distributions and load balancing
  2. **Data-level parallelism** – Utilizing the capabilities of AVX-512 ISA
- Achieve 2.9x speedup in the performance of the flux kernel relative to the baseline code
- Exhibit almost linear scalability up to the full core count of KNL (64 cores), and continued scalability with SMT
- Maintain on Skylake roughly similar speedup with less power consumption [KNL: 245 Watts; SKX: 330 Watts]

Future Direction $\longrightarrow$ Port PETSc-FUN3D onto GPU

Q/A