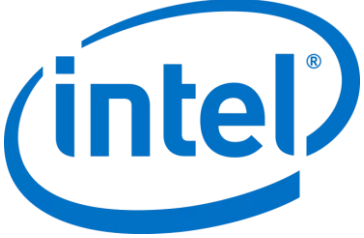


Improving Oil and Gas Extraction Simulation Performance using Intel Xeon and Xeon Phi Architectures

Matheus Serpa, Eduardo Cruz and Philippe Navaux

Informatics Institute
Federal University of Rio Grande do Sul (UFRGS)

Hillsboro, September 27

IXPUG Annual Fall Conference 2018

Goal of this work

Improve an oil and gas exploration application performance using Intel Xeon and Xeon Phi.

Methodology

Application

- Oil and Gas exploration application provided by Petrobras
  - Cooperation project ITA-UFRGS-PETROBRAS
    - Coordinated by Prof. Philippe O. A. Navaux
    - Performance evaluation and optimization of geophysics models

- Application
  - 3D Wave Propagation Model
  - Written in C + OpenMP

Methodology

Platforms

- Evaluation in 2 real machines
  - Broadwell
    - 2 x Intel Xeon E5-2699 v4 (22 cores, 2-SMT, 256-bit VPU)
    - 32 KB L1, 256 KB L2, 55 MB shared L3
    - 88 threads
    - Q1’16

Methodology

Platforms

- Evaluation in 2 real machines
  - Broadwell
    - 2 x Intel Xeon E5-2699 v4 (22 cores, 2-SMT, 256-bit VPU)
    - 32 KB L1, 256 KB L2, 55 MB shared L3
    - 88 threads
    - Q1’16
  - Knights Landing
    - Intel Xeon Phi 7250 (68 cores, 4-SMT, 512-bit VPU)
    - 32 KB L1, 512 KB L2
    - 272 threads
    - Q2’16

Optimization Strategies

Memory Access Pattern to Improve Data Locality

Optimization Strategies

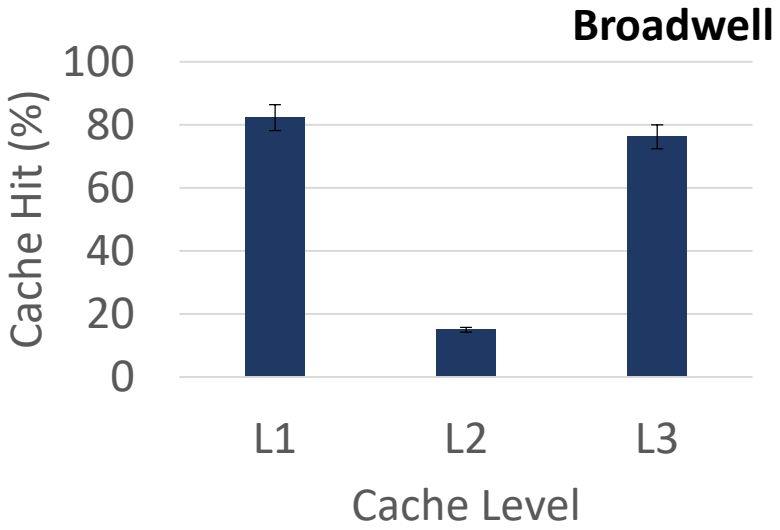
Memory Access Pattern to Improve Data Locality

- Cache performance

Optimization Strategies

Memory Access Pattern to Improve Data Locality

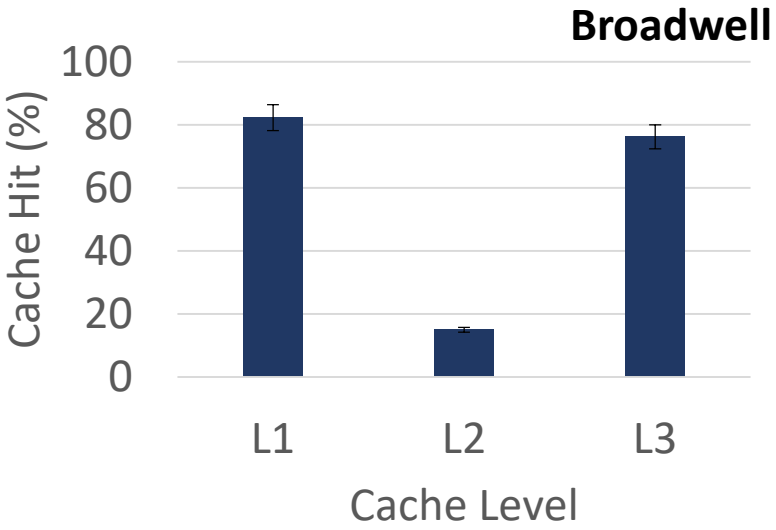
- Cache performance

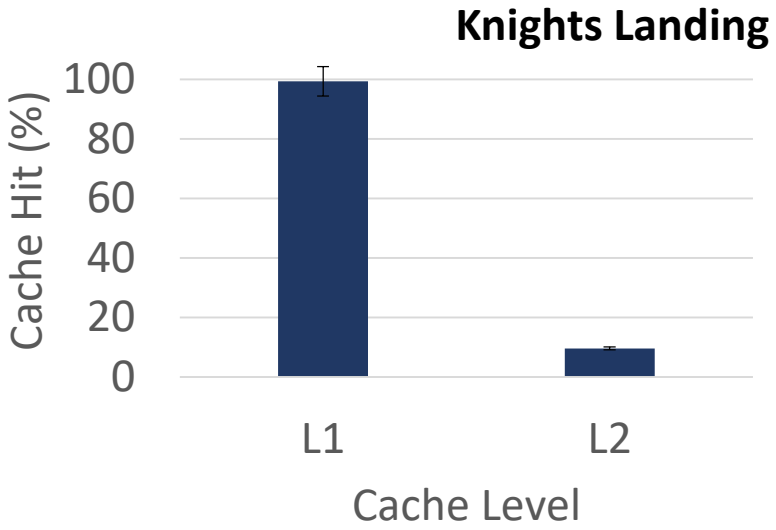


Optimization Strategies

Memory Access Pattern to Improve Data Locality

- Cache performance





Optimization Strategies

Memory Access Pattern to Improve Data Locality

```
for(x = 0; x < nnoi; x++)
    for(y = 0; y < nnoj; y++)
        for(z = 0; z < k1 - k0; z++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
```

Application main loop

Optimization Strategies

Memory Access Pattern to Improve Data Locality

```
for(x = 0; x < nnoi; x++)
    for(y = 0; y < nnoj; y++)
        for(z = 0; z < k1 - k0; z++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
```

Application main loop

index
1572864
1835008
2097152
2359296

Optimization Strategies

Memory Access Pattern to Improve Data Locality

```
for(x = 0; x < nnoi; x++)
    for(y = 0; y < nnoj; y++)
        for(z = 0; z < k1 - k0; z++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
```

Application main loop

index	stride
1572864	262144
1835008	262144
2097152	262144
2359296	

Optimization Strategies

Memory Access Pattern to Improve Data Locality

```
for(x = 0; x < nnoi; x++)
    for(y = 0; y < nnoj; y++)
        for(z = 0; z < k1 - k0; z++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
```

Application main loop

index	stride
1572864	262144
1835008	262144
2097152	262144
2359296	

- Spatial locality in caches
  - If a memory position is referenced, then it is likely that nearby memory positions will be referenced in the near future.
- Loop interchange to improve data locality
  - exchanging the order of two or more loops we reduce the stride size

Optimization Strategies

Memory Access Pattern to Improve Data Locality

```
for(x = 0; x < nnoi; x++)
    for(y = 0; y < nnoj; y++)
        for(z = 0; z < k1 - k0; z++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
            Application main loop
```

Loop Interchange

```
for(y = 0; y < nnoj; y++)
    for(z = 0; z < k1 – k0; z++)
        for(x = 0; x < nnoi; x++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
            Application main loop
```

index	stride
1572864	262144
1835008	262144
2097152	262144
2359296	

Optimization Strategies

Memory Access Pattern to Improve Data Locality

```
for(x = 0; x < nnoi; x++)
    for(y = 0; y < nnoj; y++)
        for(z = 0; z < k1 - k0; z++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
            Application main loop
```

index	stride
1572864	262144
1835008	262144
2097152	262144
2359296	

Loop Interchange

```
for(y = 0; y < nnoj; y++)
    for(z = 0; z < k1 – k0; z++)
        for(x = 0; x < nnoi; x++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
            Application main loop
```

index
1572864
1572865
1572866
1572867

Optimization Strategies

Memory Access Pattern to Improve Data Locality

```
for(x = 0; x < nnoi; x++)
    for(y = 0; y < nnoj; y++)
        for(z = 0; z < k1 - k0; z++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
            Application main loop
```

index	stride
1572864	262144
1835008	262144
2097152	262144
2359296	

Loop Interchange

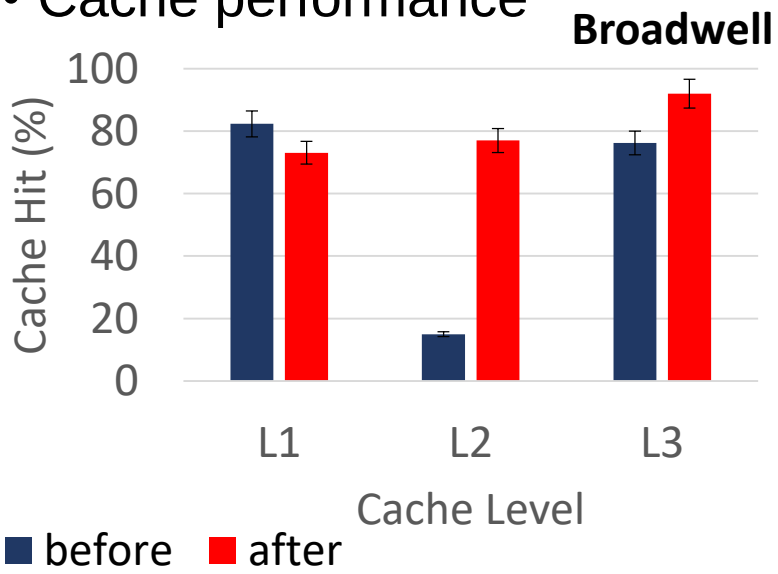
```
for(y = 0; y < nnoj; y++)
    for(z = 0; z < k1 – k0; z++)
        for(x = 0; x < nnoi; x++)
            index = (y * nnoi + x) + (k0 + z) * arm;
            …
            Application main loop
```

index	stride
1572864	1
1572865	1
1572866	1
1572867	

Optimization Strategies

Memory Access Pattern to Improve Data Locality

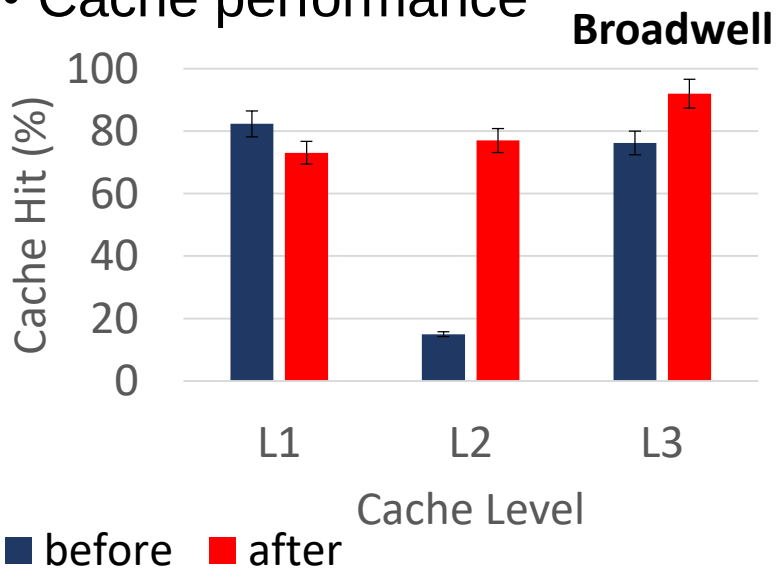
- Cache performance



Optimization Strategies

Memory Access Pattern to Improve Data Locality

- Cache performance

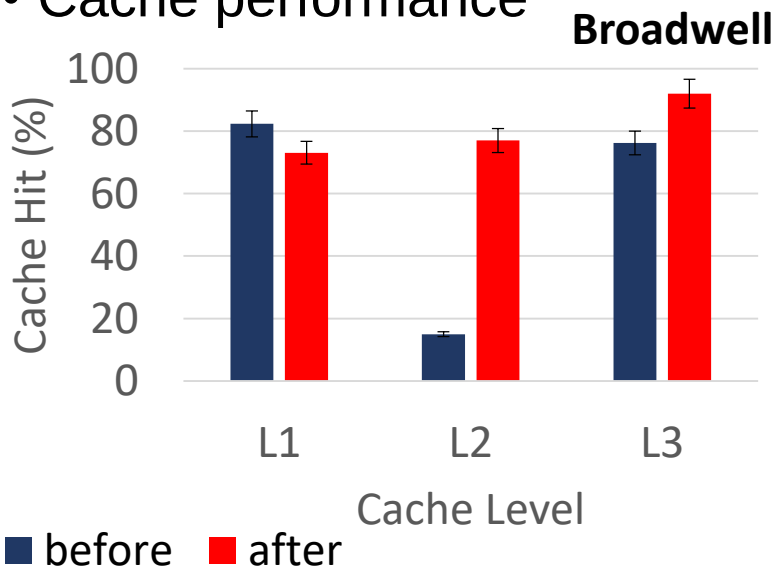


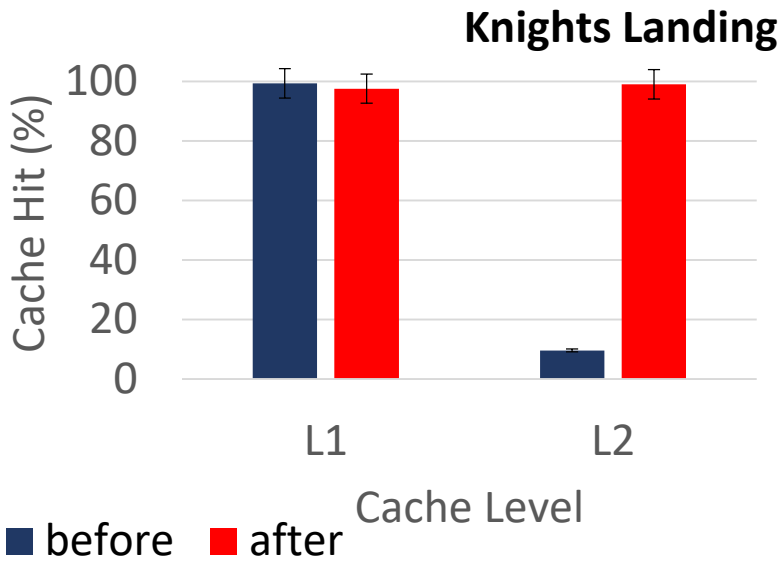
+62% +15.8%

Optimization Strategies

Memory Access Pattern to Improve Data Locality

- Cache performance



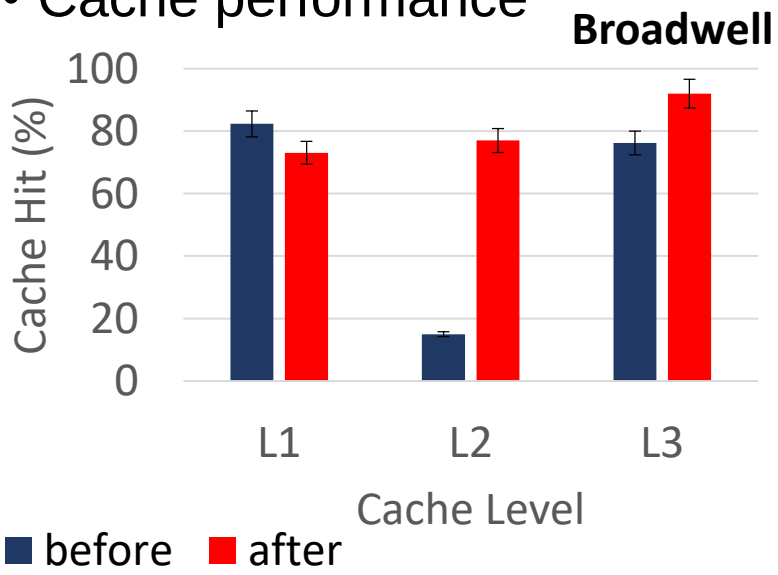


+62% +15.8%

Optimization Strategies

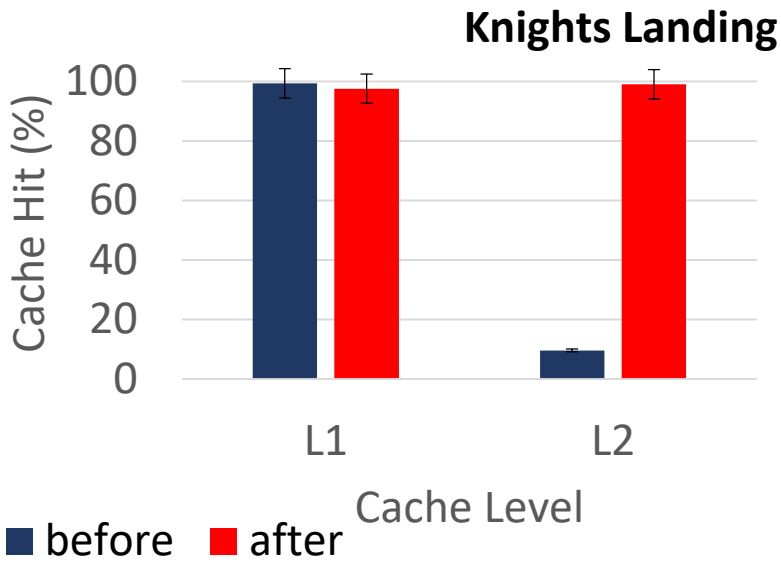
Memory Access Pattern to Improve Data Locality

- Cache performance



+62%

+15.8%

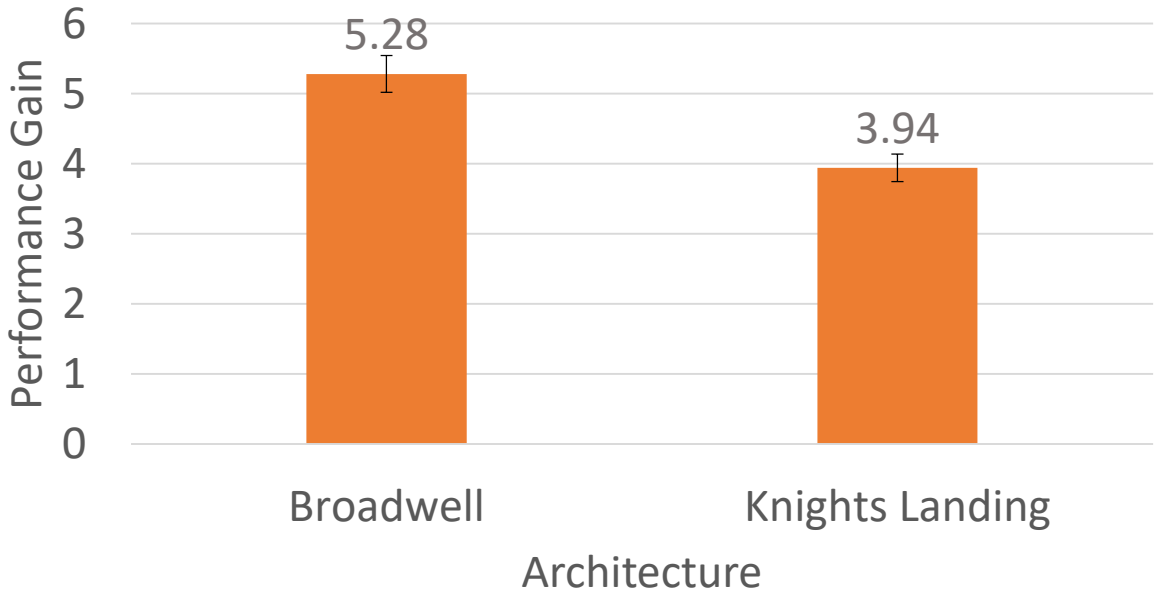


+89.5%

Optimization Strategies

Memory Access Pattern to Improve Data Locality

- Performance gain
  - Over xyz



Optimization Strategies

Exploiting SIMD for Floating-point Computations

Optimization Strategies

Exploiting SIMD for Floating-point Computations

- Many compilers feature auto-vectorization
  - However, some vectorizations cannot be fully checked at compile time

Optimization Strategies

Exploiting SIMD for Floating-point Computations

- Many compilers feature auto-vectorization
  - However, some vectorizations cannot be fully checked at compile time
- We looked at the number of AVX instructions
  - Compiler remarks

Optimization Strategies

Exploiting SIMD for Floating-point Computations

- Many compilers feature auto-vectorization
  - However, some vectorizations cannot be fully checked at compile time
- We looked at the number of AVX instructions
  - Compiler remarks
- Manual vectorization

Optimization Strategies

Exploiting SIMD for Floating-point Computations

- Many compilers feature auto-vectorization
  - However, some vectorizations cannot be fully checked at compile time
- We looked at the number of AVX instructions
  - Compiler remarks
- Manual vectorization
  - Directives to manually vectorize the code

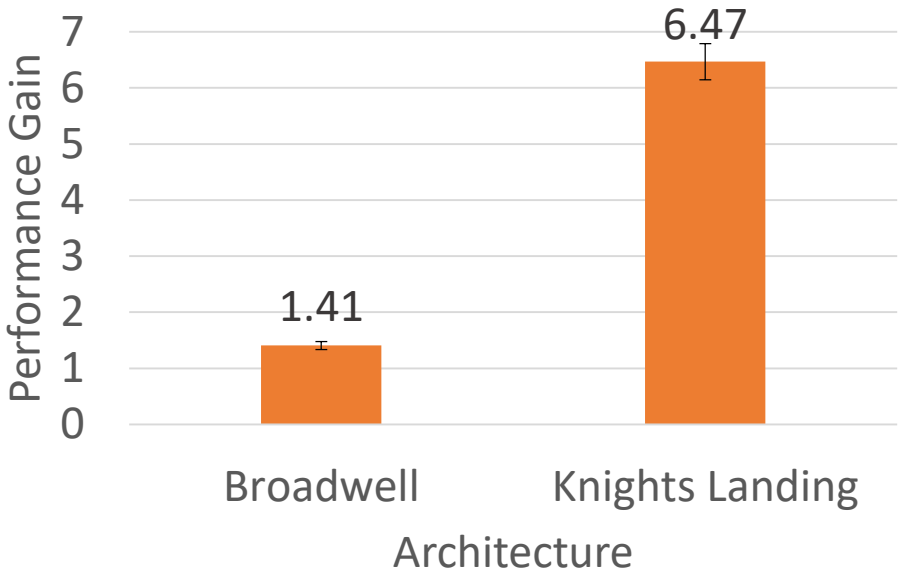
```
for(y = 0; y < nnoj; y++)
    for(z = 0; z < k1 – k0; z++)
        #pragma simd
        for(x = 0; x < nnoi; x++)
            index = (y * nnoi + x) + (k0 + z) * arm;
```

Application main loop

Optimization Strategies

Exploiting SIMD for Floating-point Computations

- Performance gain
  - Over code not vectorized



Optimization Strategies

Optimizing Memory Affinity

Optimization Strategies

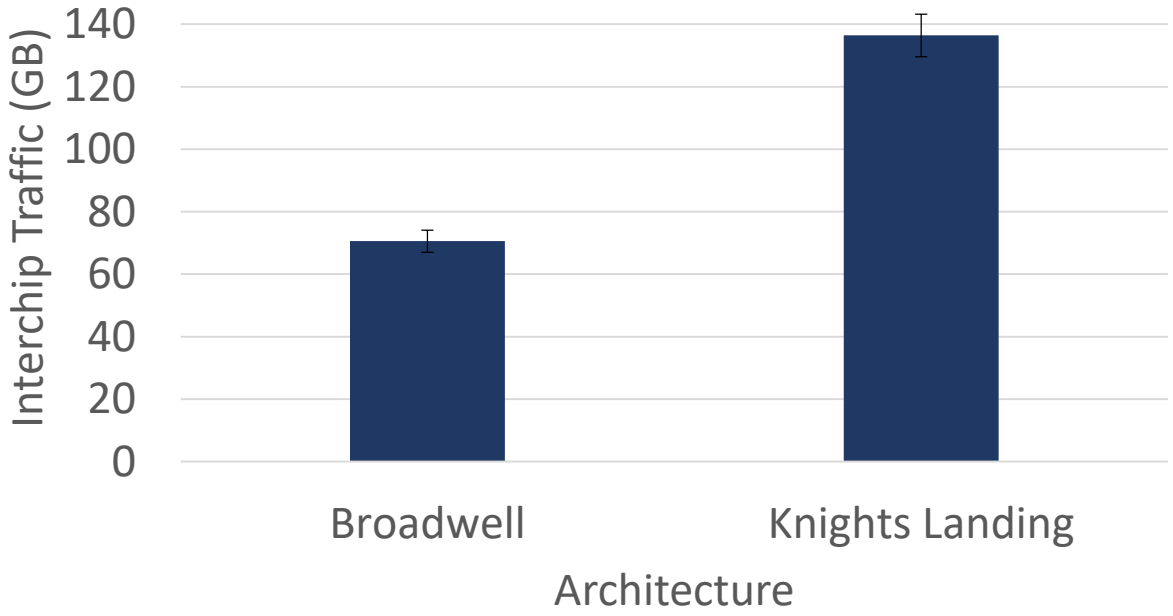
Optimizing Memory Affinity

- Interchip Traffic (GB)

Optimization Strategies

Optimizing Memory Affinity

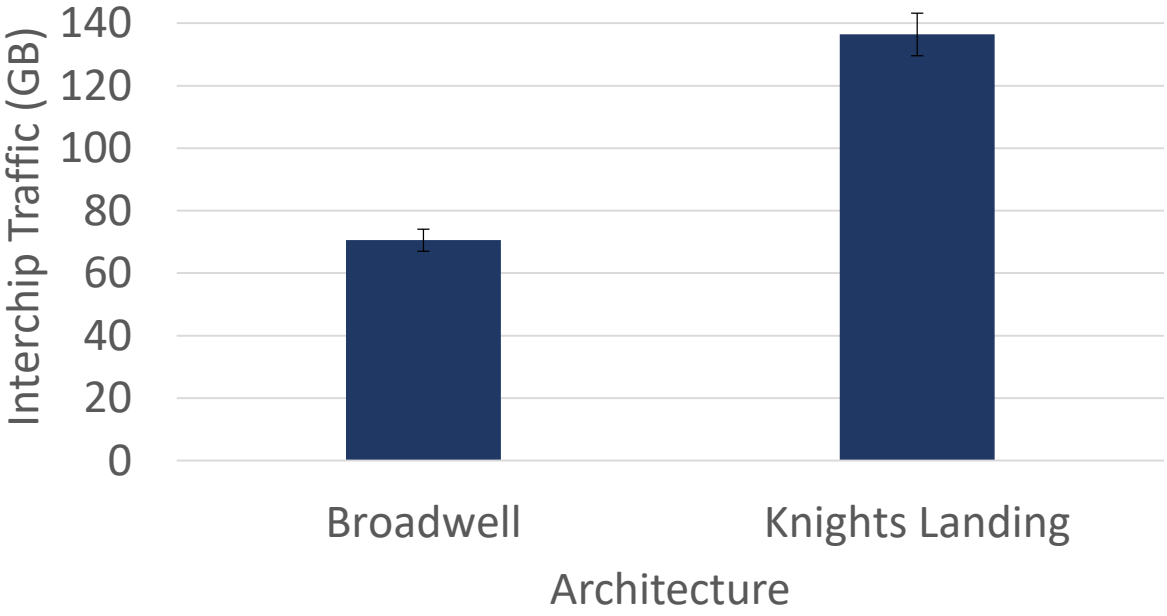
- Interchip Traffic (GB)



Optimization Strategies

Optimizing Memory Affinity

- Interchip Traffic (GB)
- Thread and data mapping to reduce latency



Optimization Strategies

Optimizing Memory Affinity

- We used thread and data mapping to reduce latency

Optimization Strategies

Optimizing Memory Affinity

- We used thread and data mapping to reduce latency

- Thread Mapping
  - Map threads that communicate to cores close in the memory hierarchy.

Optimization Strategies

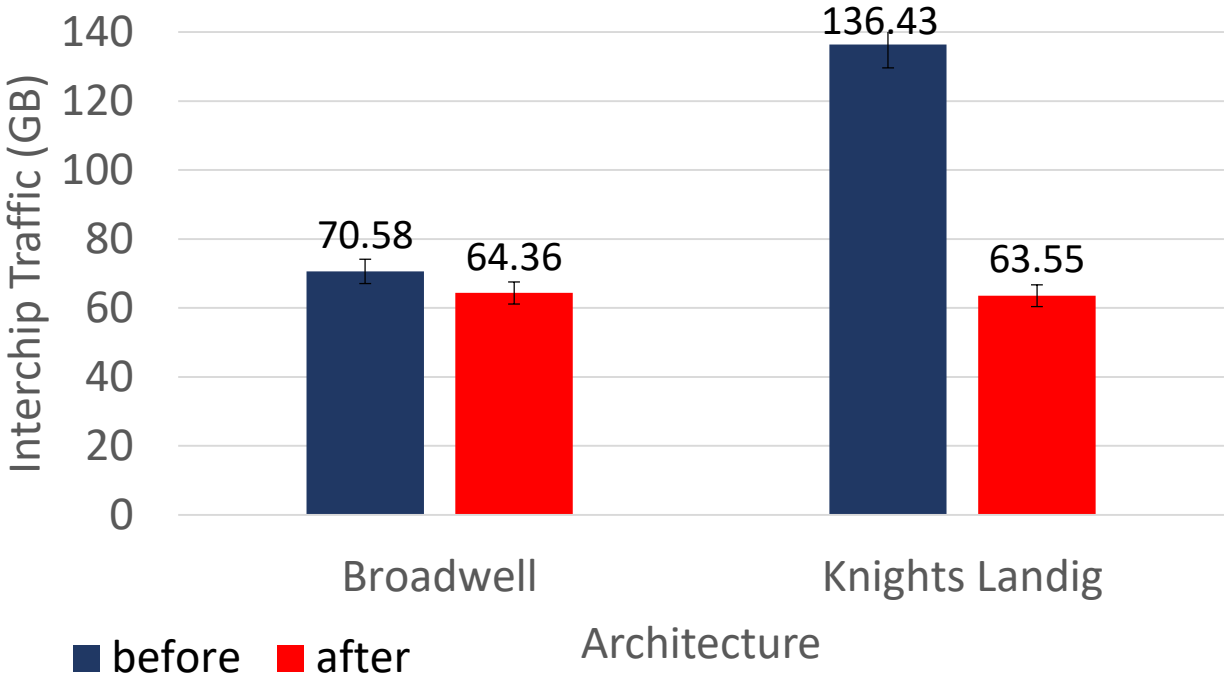
Optimizing Memory Affinity

- We used thread and data mapping to reduce latency
- Thread Mapping
  - Map threads that communicate to cores close in the memory hierarchy.
- Data Mapping
  - Map pages to the NUMA node near to the threads that performs most memory accesses to them.

Optimization Strategies

Optimizing Memory Affinity

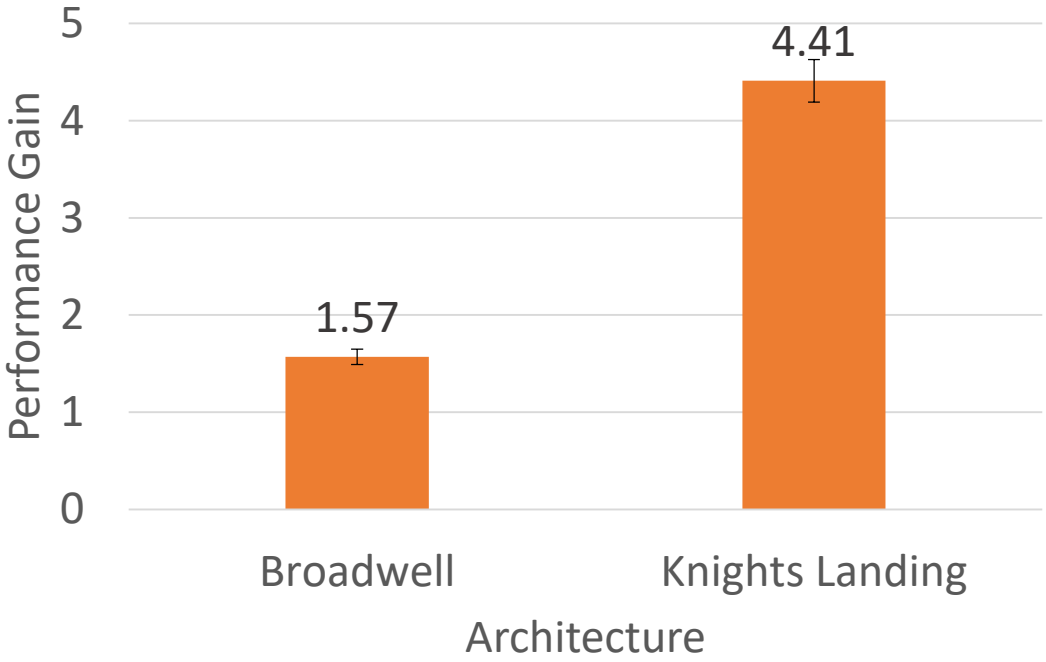
- Interchip Traffic (GB)



Optimization Strategies

Optimizing Memory Affinity

- Performance gain



Optimization Strategies

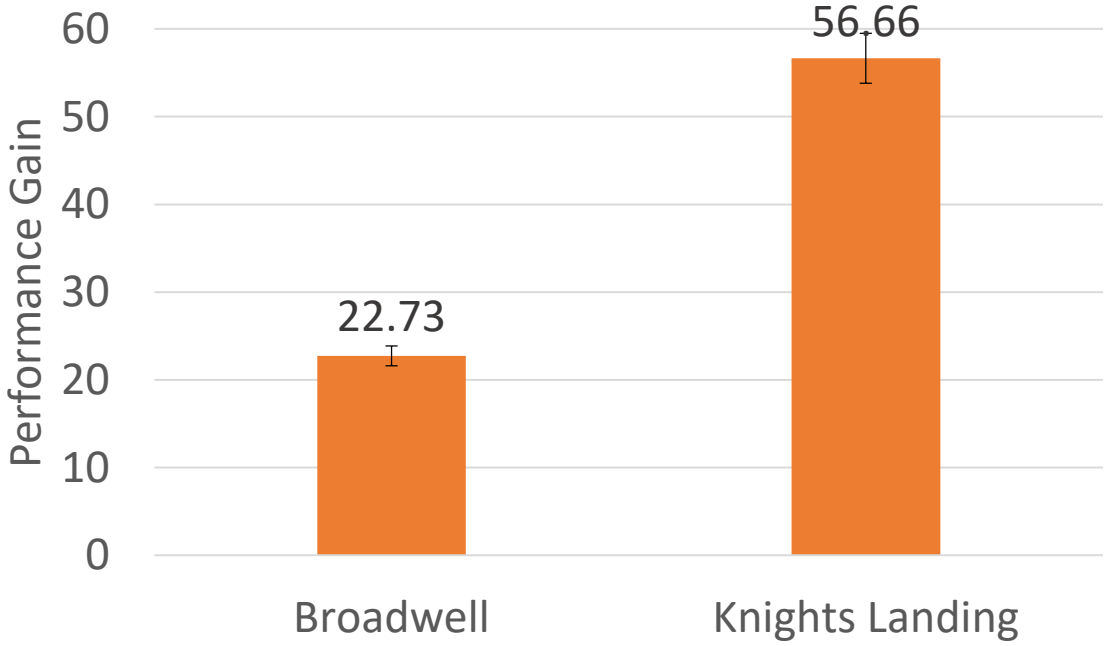
Performance Comparison

- We applied all optimizations together

Optimization Strategies

Performance Comparison

- We applied all optimizations together
- Performance over naive version



Optimization Strategies

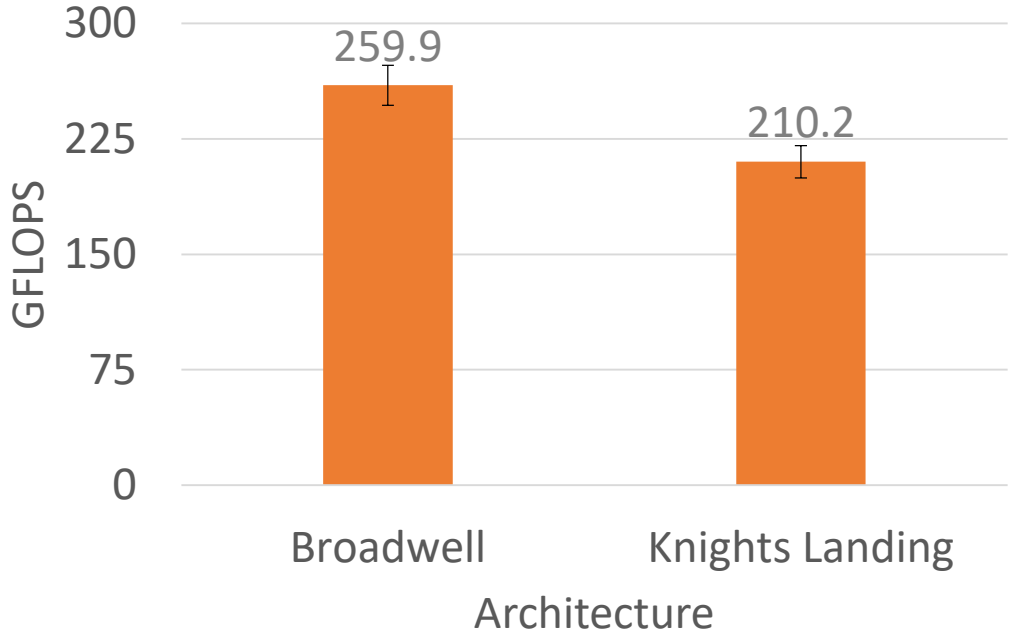
Performance Comparison

- Optimized application performance

Optimization Strategies

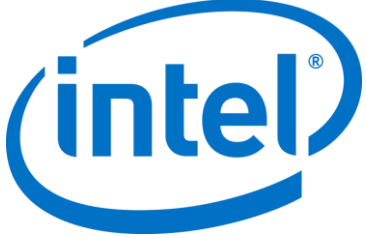
Performance Comparison

- Optimized application performance
  - 1.2x faster on Broadwell



Conclusion

- We optimize an oil and gas application and use hardware counters to measure the impact of each strategy.
  - Loop Interchange (up to 5.3 on Broadwell)
  - Explicit Vectorization (up to 6.5 on Knights Landing)
  - Thread and Data Mapping (up to 4.4 on Knights Landing)
- Broadwell performance
  - 1.2x over Knights Landing



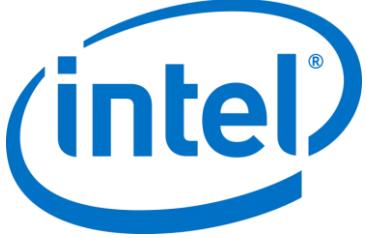
Acknowledgements

This research received funding from Intel under the Modern Code Project, and Petrobras under 2016/00133-9.

Matheus Serpa <msserpa@inf.ufrgs.br>

Hillsboro, September 27

IXPUG Annual Fall Conference 2018



Thanks!
Questions?

Matheus Serpa <msserpa@inf.ufrgs.br>

Informatics Institute
Federal University of Rio Grande do Sul (UFRGS)

Hillsboro, September 27

IXPUG Annual Fall Conference 2018