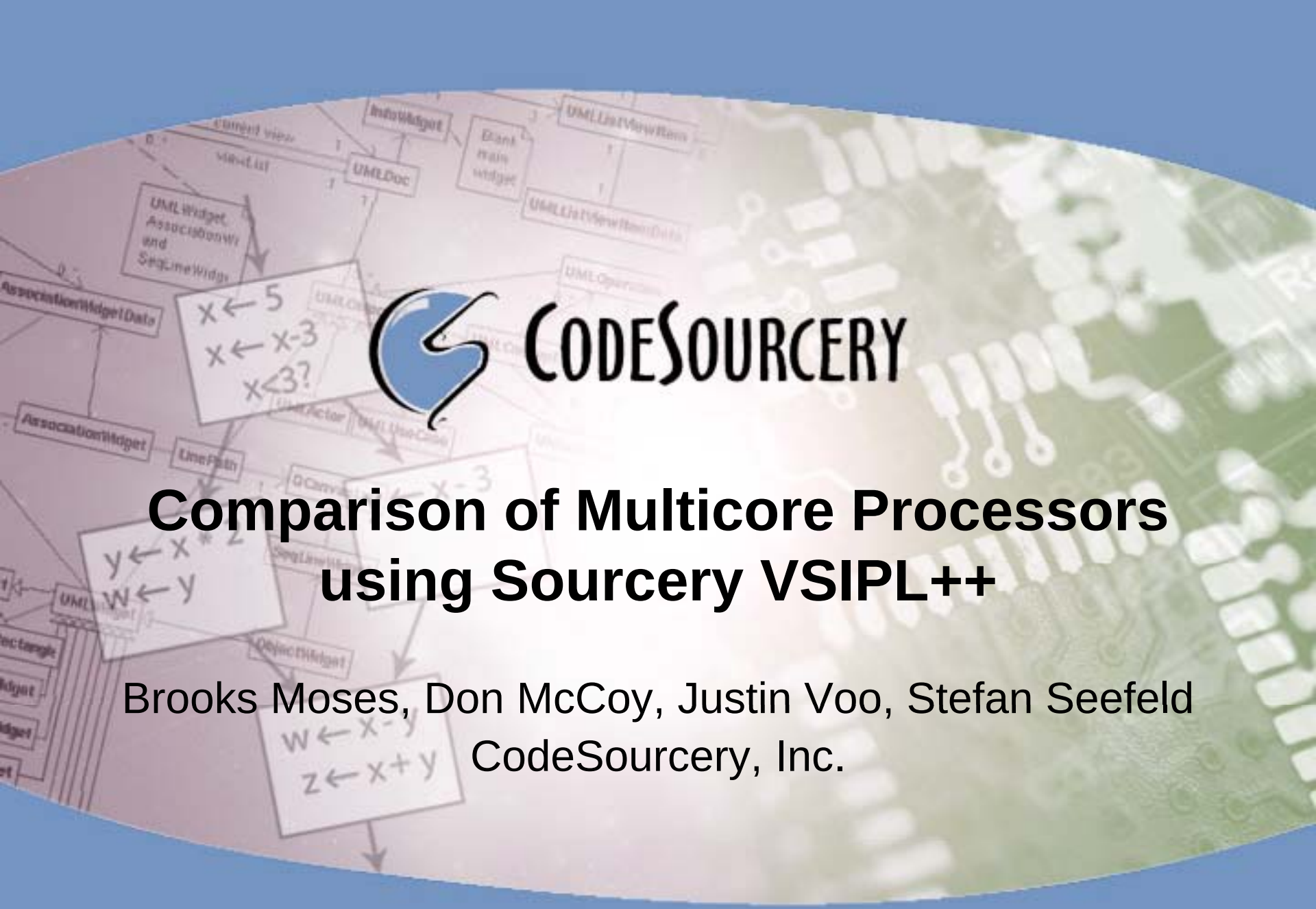


Comparison of Multicore Processors using Sourcery VSIPL++

Brooks Moses, Don McCoy, Justin Voo, Stefan Seefeld
CodeSourcery, Inc.

Outline

- Estimating Performance: Modern multicore processors have complex performance characteristics.
- Comparing Performance: x86, CUDA, and Cell/B.E. implementations of Sourcery VSIPL++ operations.
  - Elementwise functions
  - Matrix product
  - Matrix transpose
  - FFT
  - Multiple FFT
- Conclusions

Estimating Performance

- How single-core processors (ideally) work:
  - Core has a peak theoretical MFLOPS/s performance.
  - Algorithms can achieve some fraction of peak that does not vary much across processors
    - Matrix products typically achieve 99% or so.
  - If you move to a system with 2x the MFLOPS/s, you generally get a 2x performance boost.

Estimating Performance

- It would be nice if modern multicore processors worked the same way:

	Peak GFLOPS/s	4096x4096 Matrix-Multiply performance
Intel 4-core Nehalem x86	102.4	98.1
NVIDIA Tesla C1060 GPU	933	~900?

Estimating Performance

- It would be nice if modern multicore processors worked the same way:

	Peak GFLOPS/s	4096x4096 Matrix-Multiply performance
Intel 4-core Nehalem x86	102.4	98.1
NVIDIA Tesla C1060 GPU	933	326.3

- Reality, however, is quite different.
- Constrained by memory transfer limits, in this case.

Comparing Performance

- So, what do performance comparisons on modern multicore processors look like, then?
- Sourcery VSIPL++ can help:
  - Contains a range of fundamental signal-processing operations.
  - Implemented for a range of different architectures of interest (x86, Cell/B.E., and now CUDA).

Comparing Performance

Processors used for comparison:

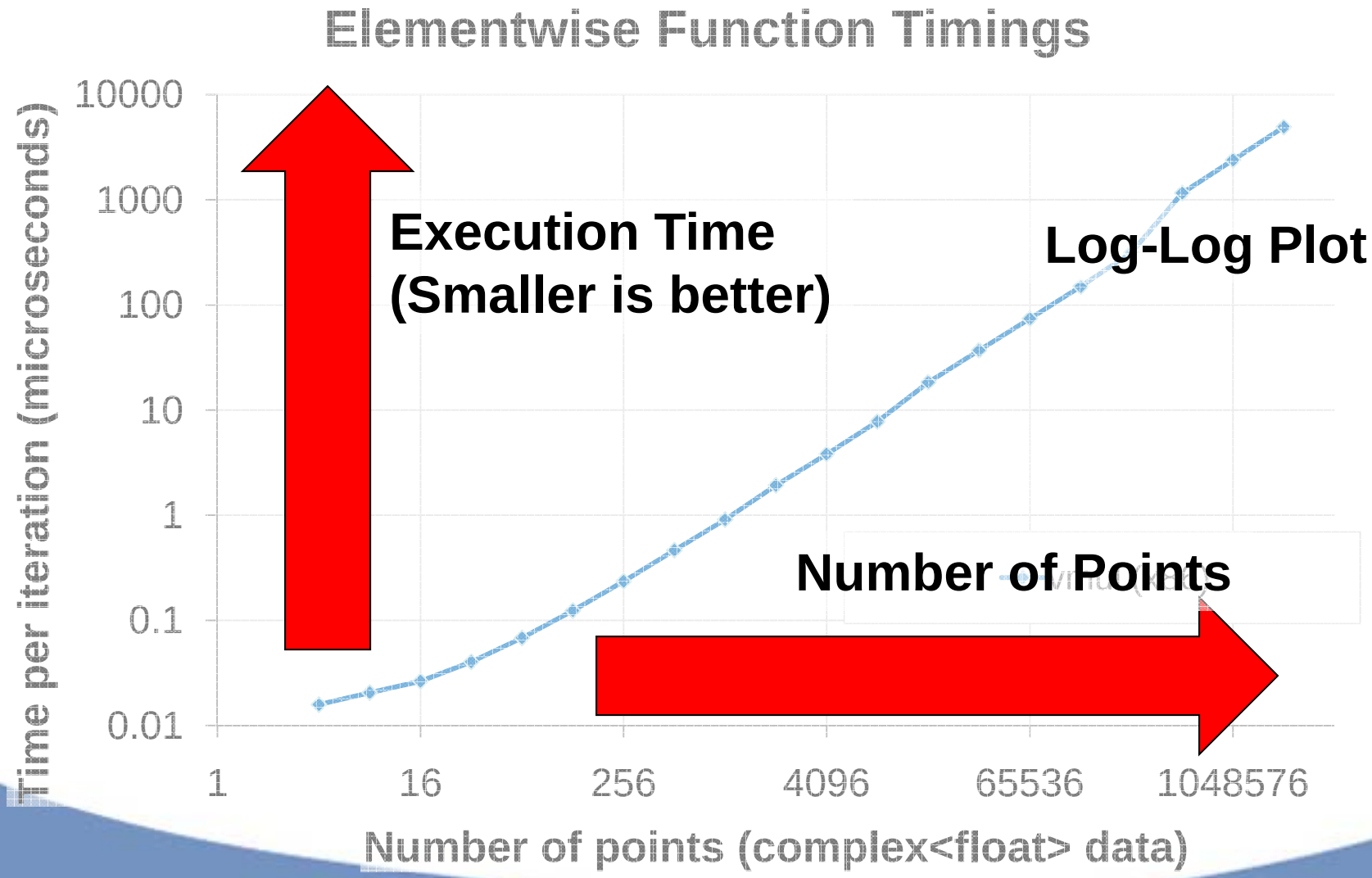
- **x86**: Intel Core i7 (Nehalem), 4 cores (hyperthreaded), 3.2 GHz, peak performance of 102.4 GFLOPS/s.
- **Cell/B.E.**: 1 Power PPE, 8 SPE coprocessors, 3.2 GHz, peak performance of 204.8 GFLOPS/s using SPEs.
- **CUDA**: NVIDIA Tesla C1060, 240 cores, 1.3 GHz, peak performance of 933 GFLOPS/s.

Elementwise Vector Operations

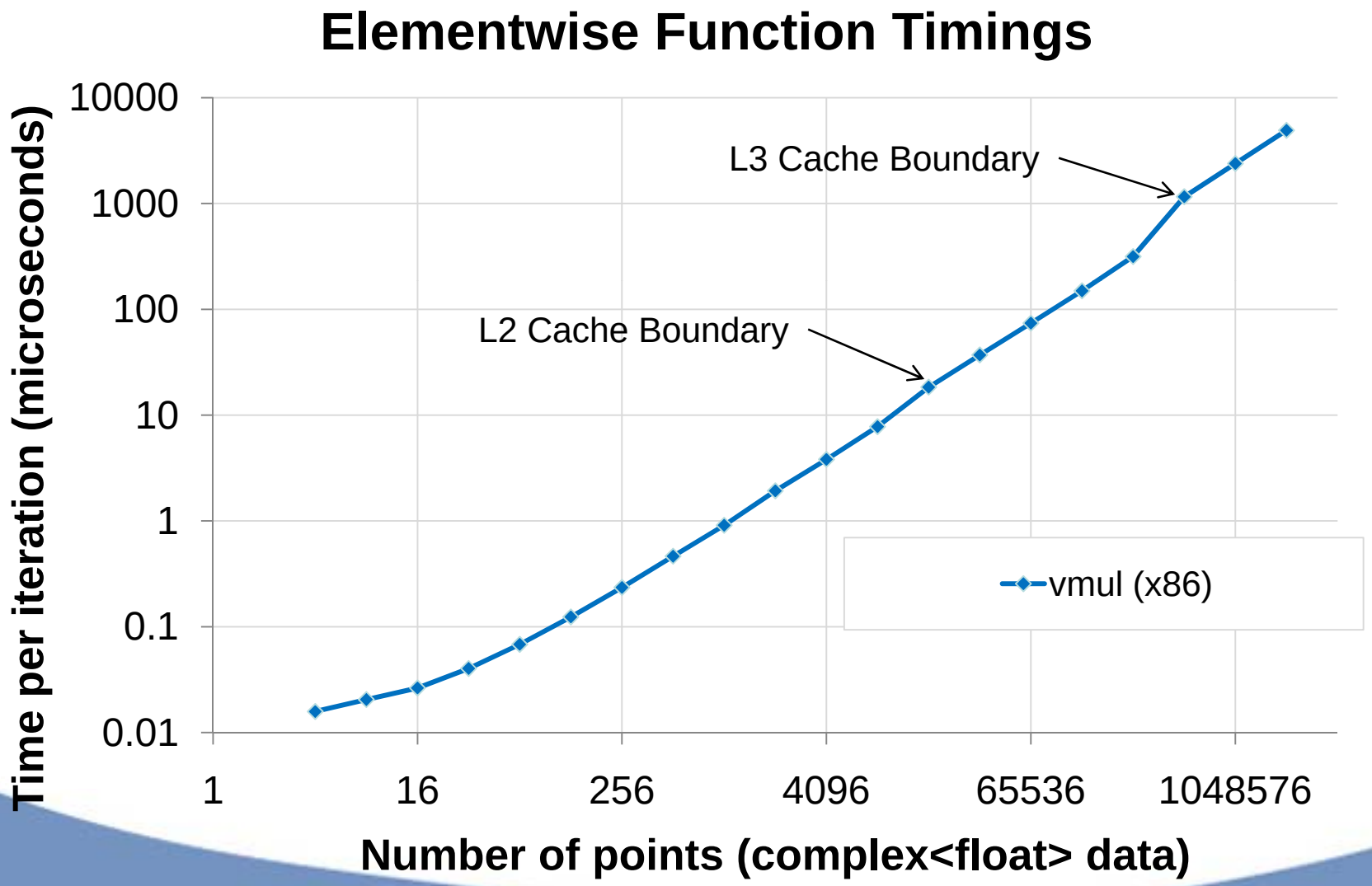
- Start with a simple operation:
  - Elementwise vector multiplication

- Memory bandwidth is a primary limiting factor.
- Illustrates some effects other than GFLOPS/s.

Elementwise Vector Operations



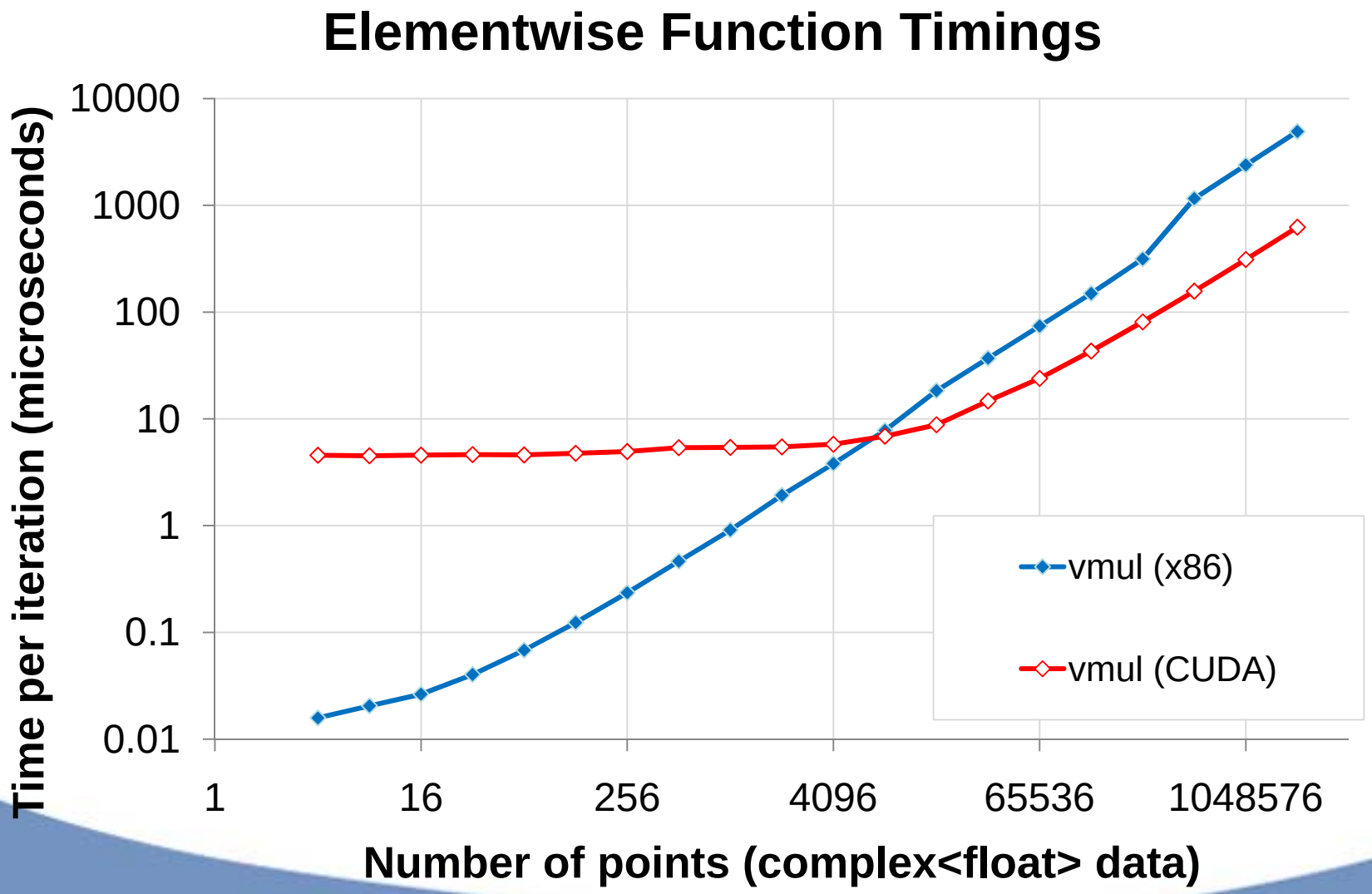
Elementwise Vector Operations



Elementwise Vector Operations

- Performance on x86:
  - Time proportional to size over a very wide range of sizes.
  - If there is a fixed cost, it is less than the cost of computing 4 elements.
  - Small performance hits when the data is too large to fit inside L2 and L3 caches.

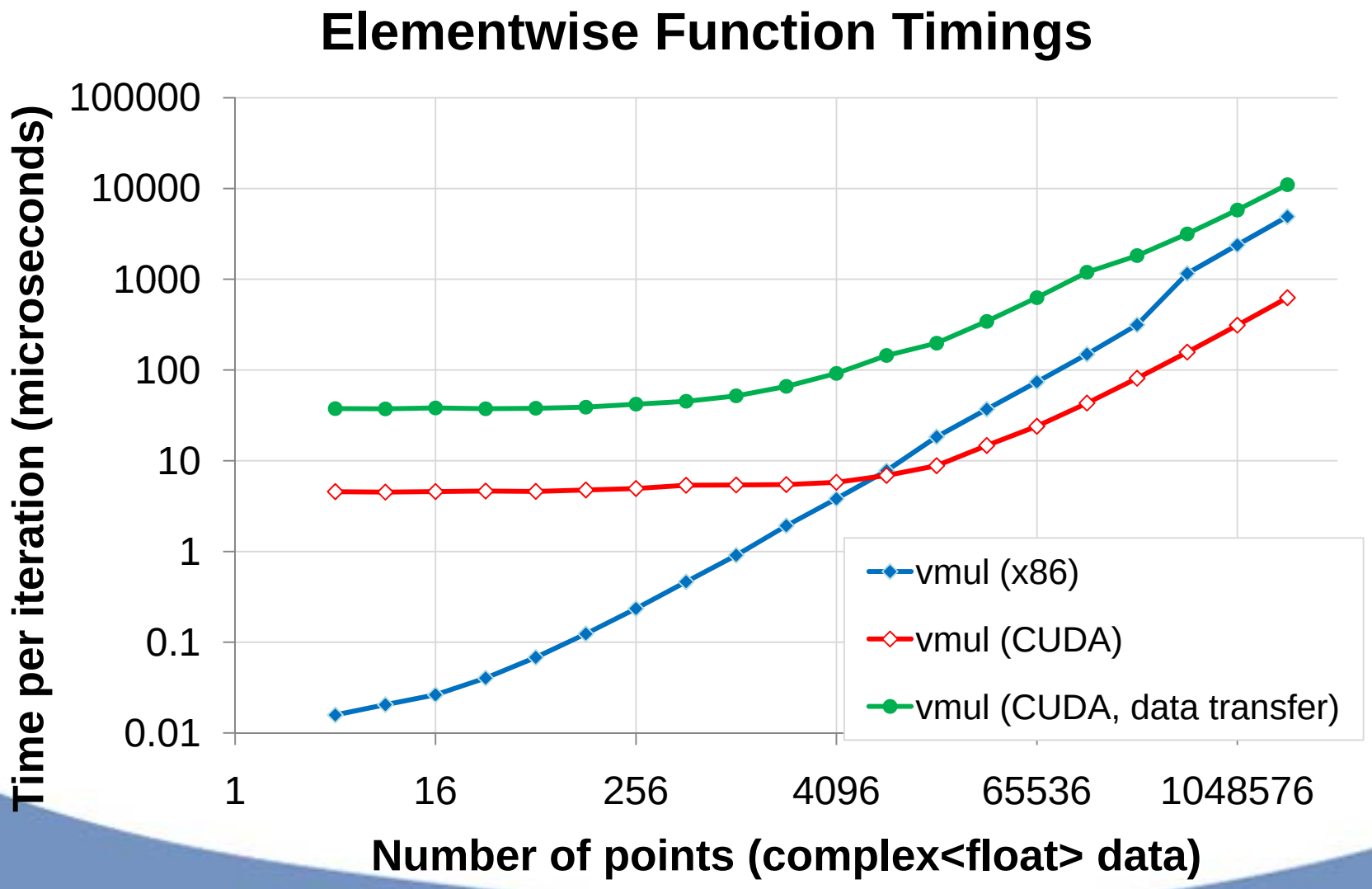
Elementwise Vector Operations



Elementwise Vector Operations

- Performance on CUDA:
  - Large fixed cost (about 5 microseconds) dominates performance for smaller data sizes.
  - Above 16k points, performance is proportional to size.
  - No memory cache hierarchy effects, as there are no memory caches.
- However, this is making an important assumption:
  - Assumes data is already in GPU memory!

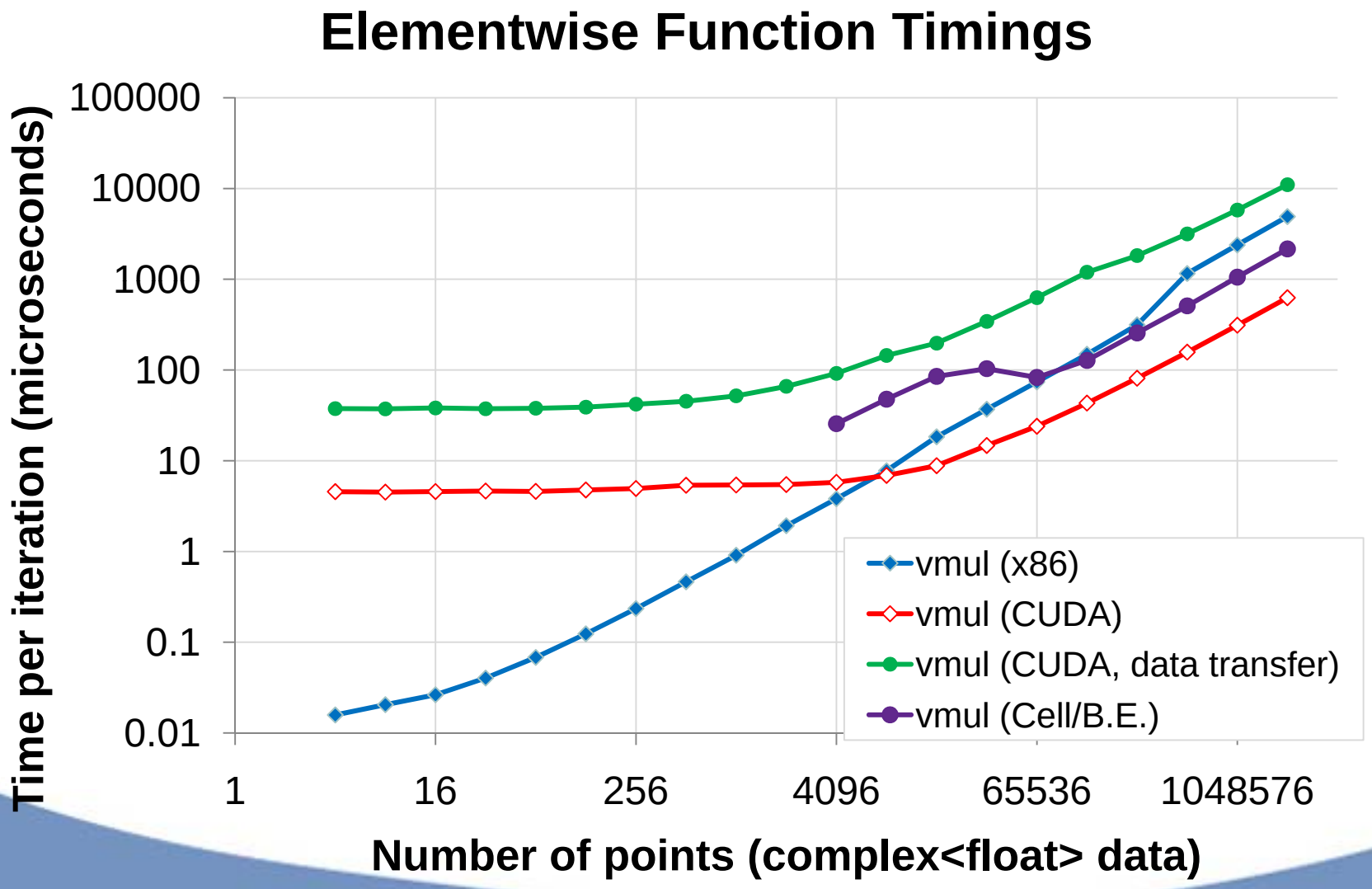
Elementwise Vector Operations



Elementwise Vector Operations

- If this data comes from system memory, and we want it to be in system memory afterwards, we need to account for data transfers.
- Adds additional fixed cost (32 microseconds) and additional cost per point.
- If all we're doing on the GPU is a single elementwise operation, the CPU is always faster – though the gap narrows to only a factor of 2 at the largest sizes.
- Key performance note: Store data on the GPU for as many sequential operations as possible.

Elementwise Vector Operations



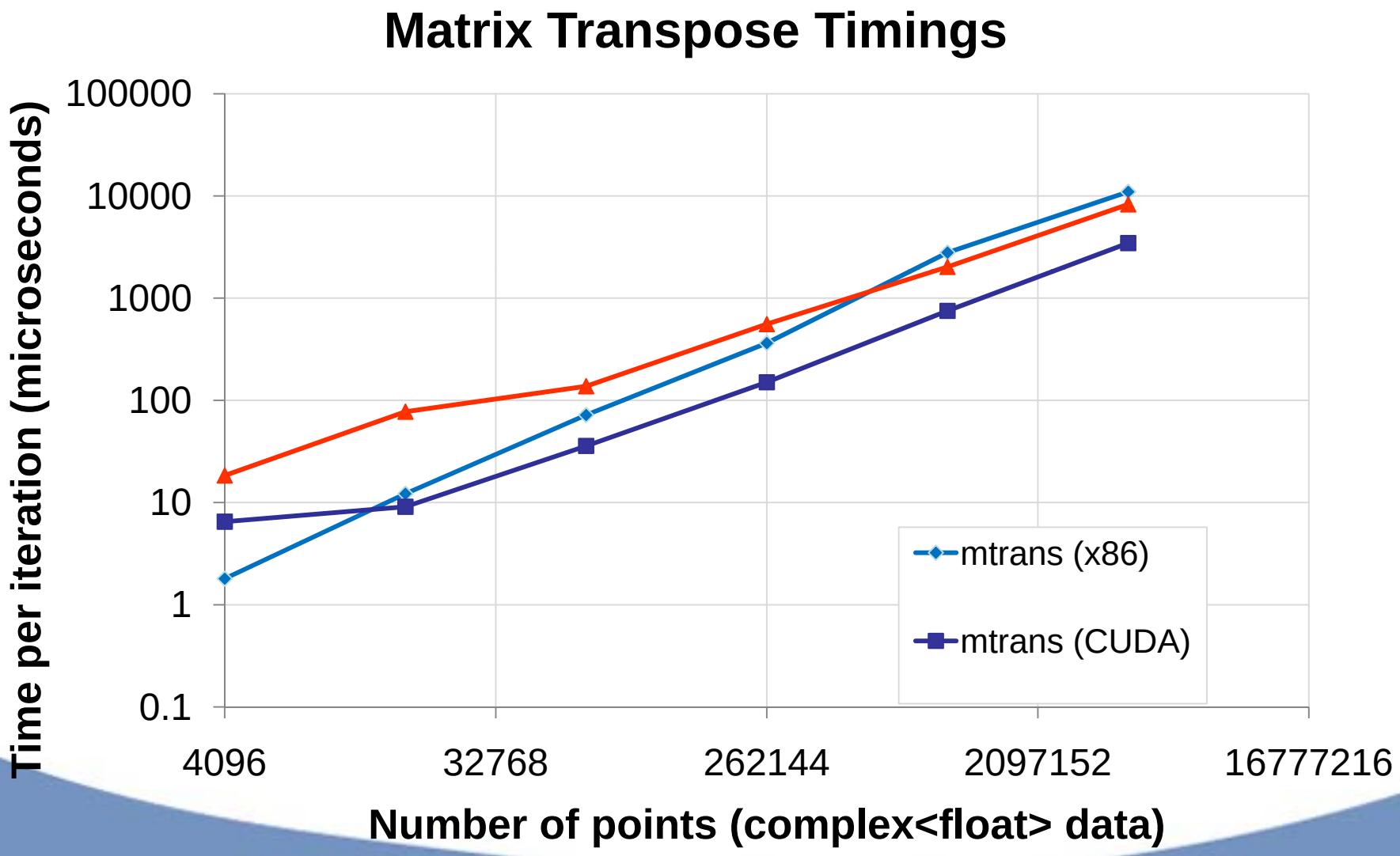
Elementwise Vector Operations

- Performance on Cell/B.E. SPEs:
  - SPEs not useful below 4k points.
  - Weird effects due to chunking problem across SPEs for smaller data sizes.
  - Data must be transferred to/from SPEs; cannot be stored in SPE local store (unlike GPU memory).
  - In general, speed is comparable to x86 L2 cache – much faster than GPU transfers from host memory, but slower than GPU device memory.

Matrix Transposition

- What about more complex data access patterns?
- Matrix transpose
  - x86: Can use 4x4 SIMD blocks, but access has large strides otherwise.
  - CUDA: Access has large strides; regularity may be a handicap due to bank conflicts.
  - Cell/B.E.: Uses block access patterns to transpose 32x32 blocks.

Matrix Transposition



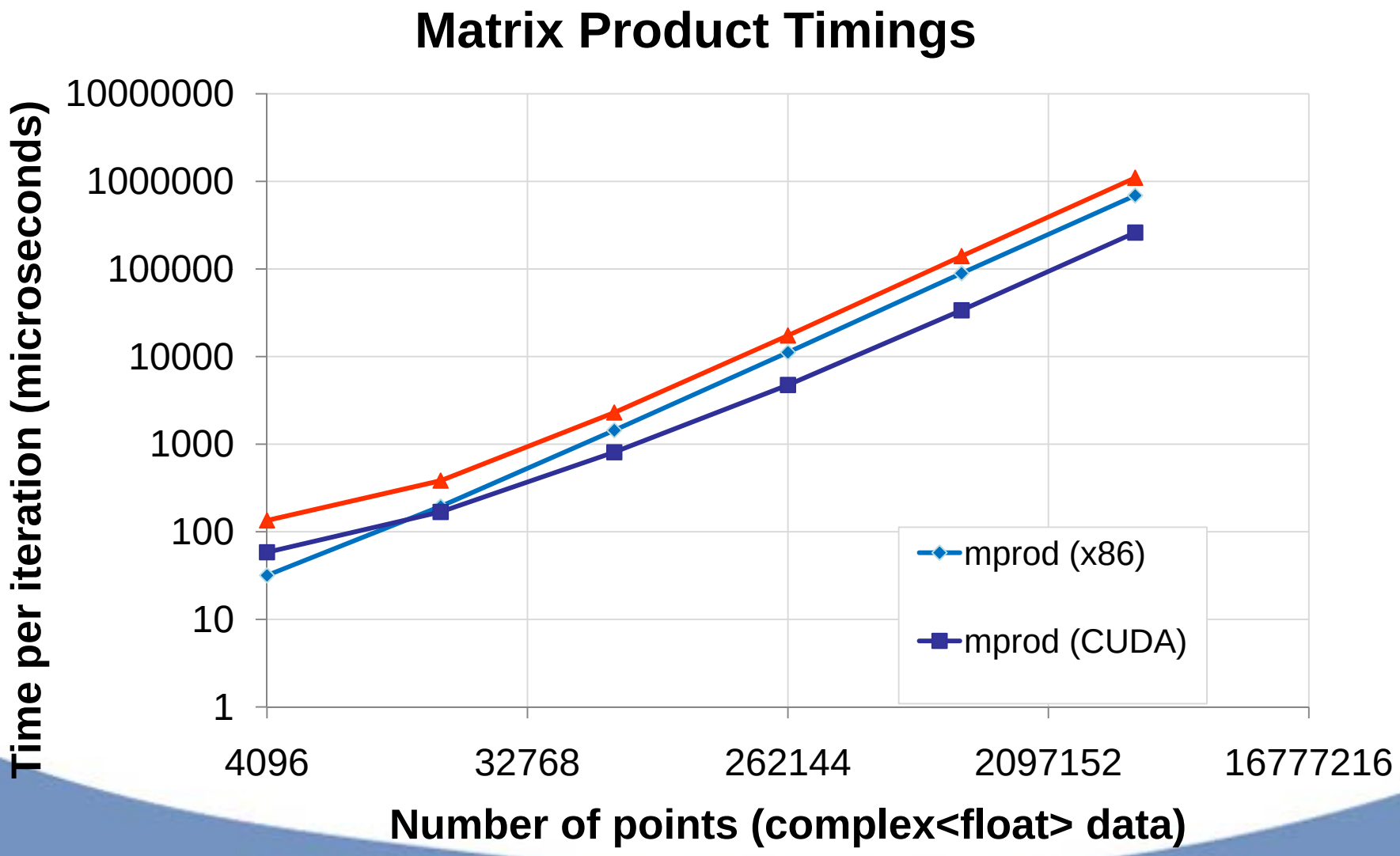
Matrix Transposition

- Results are very similar to elementwise-operation results.
  - Quantitatively, performance gap between x86 and CUDA is about 3x rather than 8x.
  - Conclusion: CUDA is slowed down somewhat by “random” data access, but is still much faster than the x86.
  - Cell performance is comparable, due to using a block strategy for data access.

Matrix Product

- So, what about something that's usually considered to be compute-bound?
- Matrix product:
  - O(N^3) operations for O(N^2) data points.
  - Implemented in block form to reduce redundant memory load costs.

Matrix Product



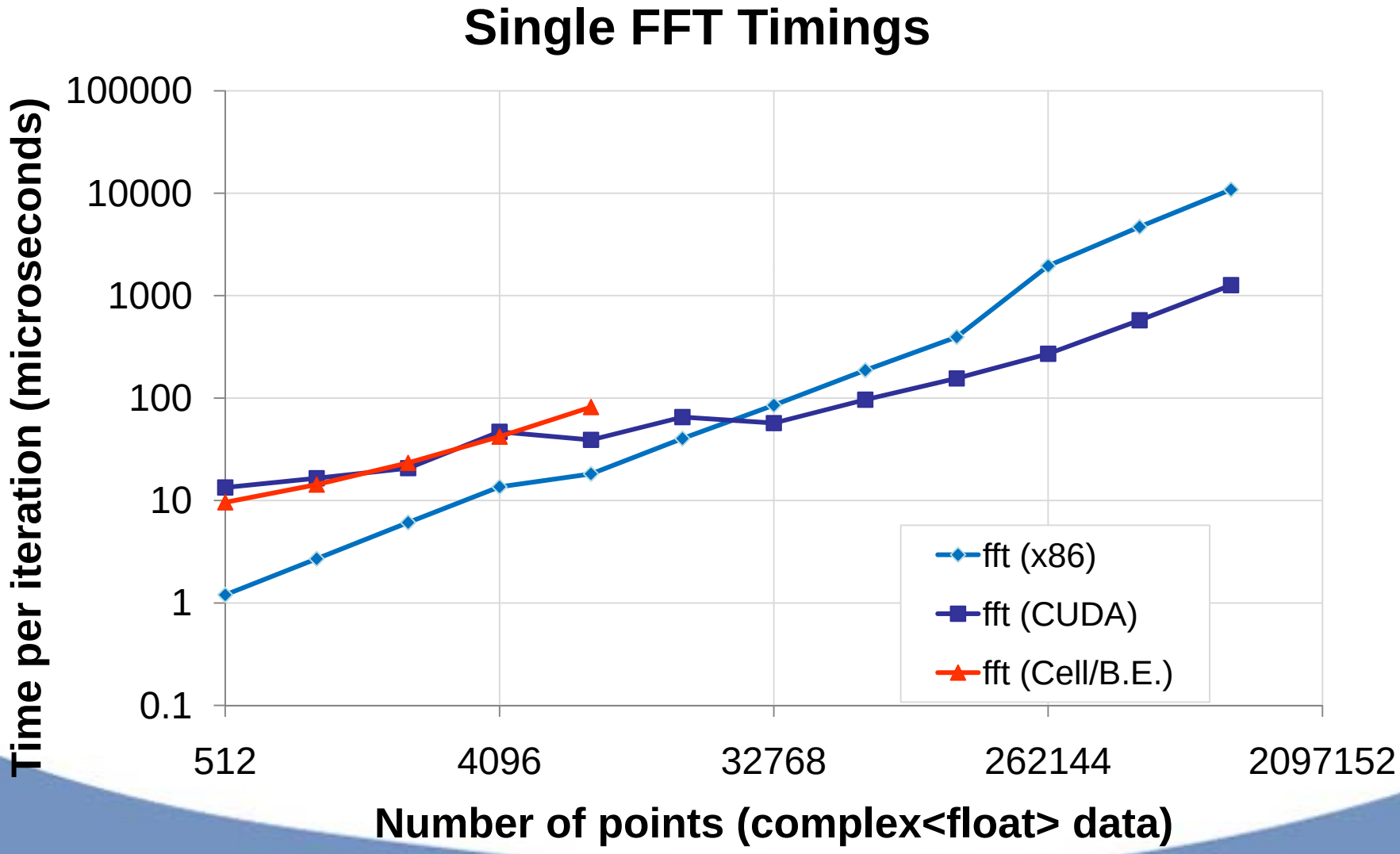
Matrix Product

- Again, similar results to matrix transpose and elementwise functions, although slope is different.
  - Results very consistent for larger data sizes (512x512 and above)
  - Cell/B.E. is 60% of x86 speed.
  - CUDA is 2.4x x86 speed.
  - Conclusion: This is not compute-bound on Cell/B.E. or CUDA.

Fast Fourier Transform

- Fast Fourier Transform (FFT):
  - Very common signal-processing operation.
  - Each element depends on other elements, requiring some form of inter-process communication or staged execution.
  - Data access is strided, with different strides at each stage.

Fast Fourier Transform



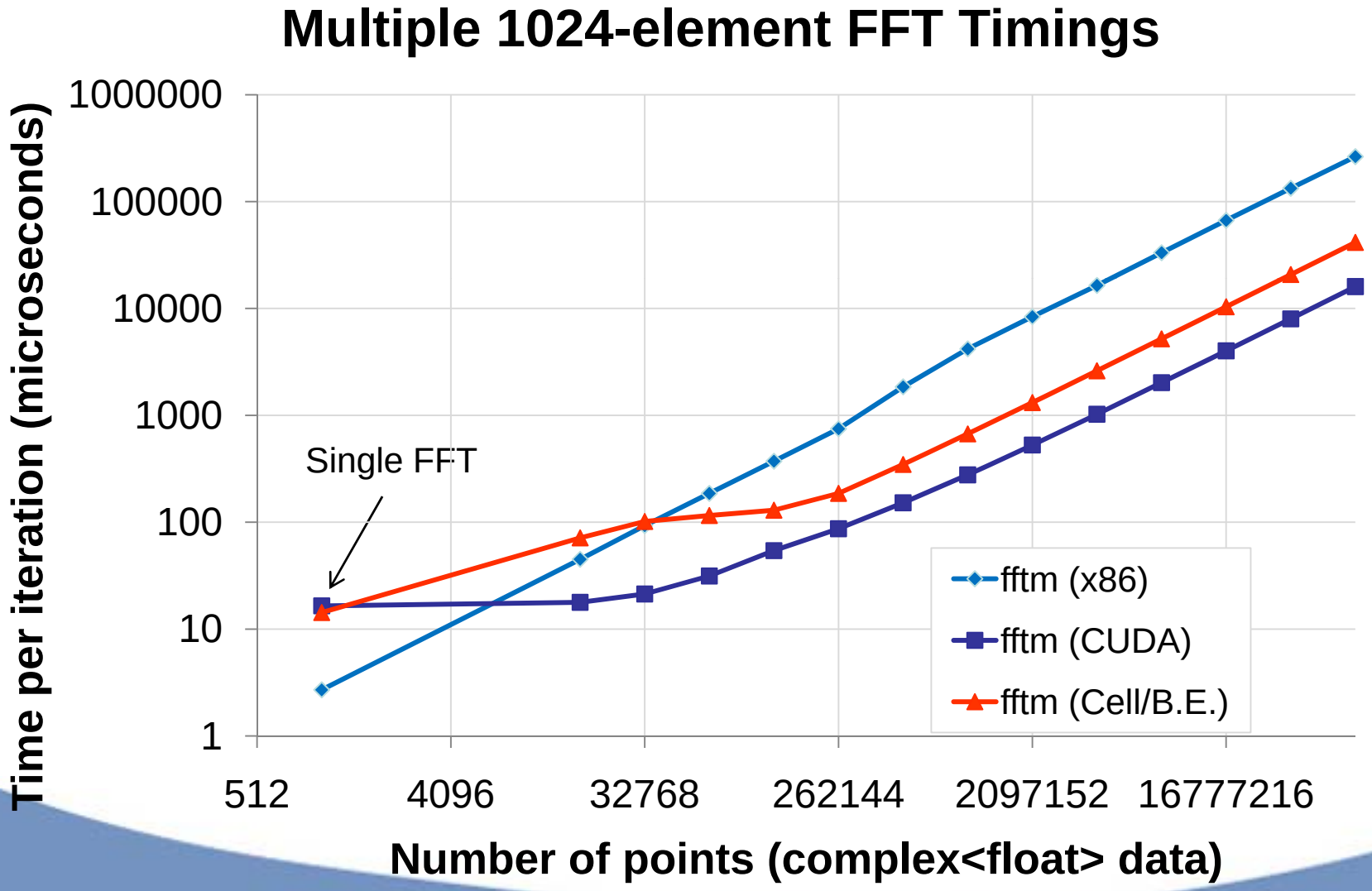
Fast Fourier Transform

- Fast Fourier Transform (FFT):
  - Again, similar x86 and CUDA patterns
  - CUDA implementation is able to account for inter-element communication efficiently, even though threads can only communicate through device memory.
  - Cell/B.E. implementation is limited to 8k data sizes, as inter-SPE communication patterns are specific to a given FFT size and difficult to generalize.

Fast Fourier Transform

- What about multiple FFTs?
  - An FFT on each row of a 1024-by-N matrix.
  - Should allow an extra level of “easy” parallelism.
  - Represents a case with a granularity in the ideal spot for Cell/B.E. SPE computations.

Fast Fourier Transform



Fast Fourier Transform

- Multiple-FFT Results:
  - x86 and CUDA results are very similar to the single-FFT case.
  - Cell/B.E. results are better (indicating that this is, indeed, a better-suited problem for the architecture), but CUDA is still significantly faster (2x-3x).

Conclusions

- Qualitatively, these graphs are all similar:
  - Most fundamental signal-processing operations can be written as reasonably ideal parallel algorithms.
  - Differences in data access patterns, inter-element communication have small effects.
- Memory transfer costs and fixed costs have a significant effect on performance of coprocessor-based systems.
  - To get good results on CUDA, the data must reside in GPU device memory for several operations.

Conclusions

- Cell/B.E. is no longer winning at any size range.
- x86 is significantly faster for operations on smaller data sizes.
- CUDA is faster (usually 3x-10x) for operations on larger data sizes.
- As a rough heuristic, the x86/CUDA crossover point tends to be around 10k points.