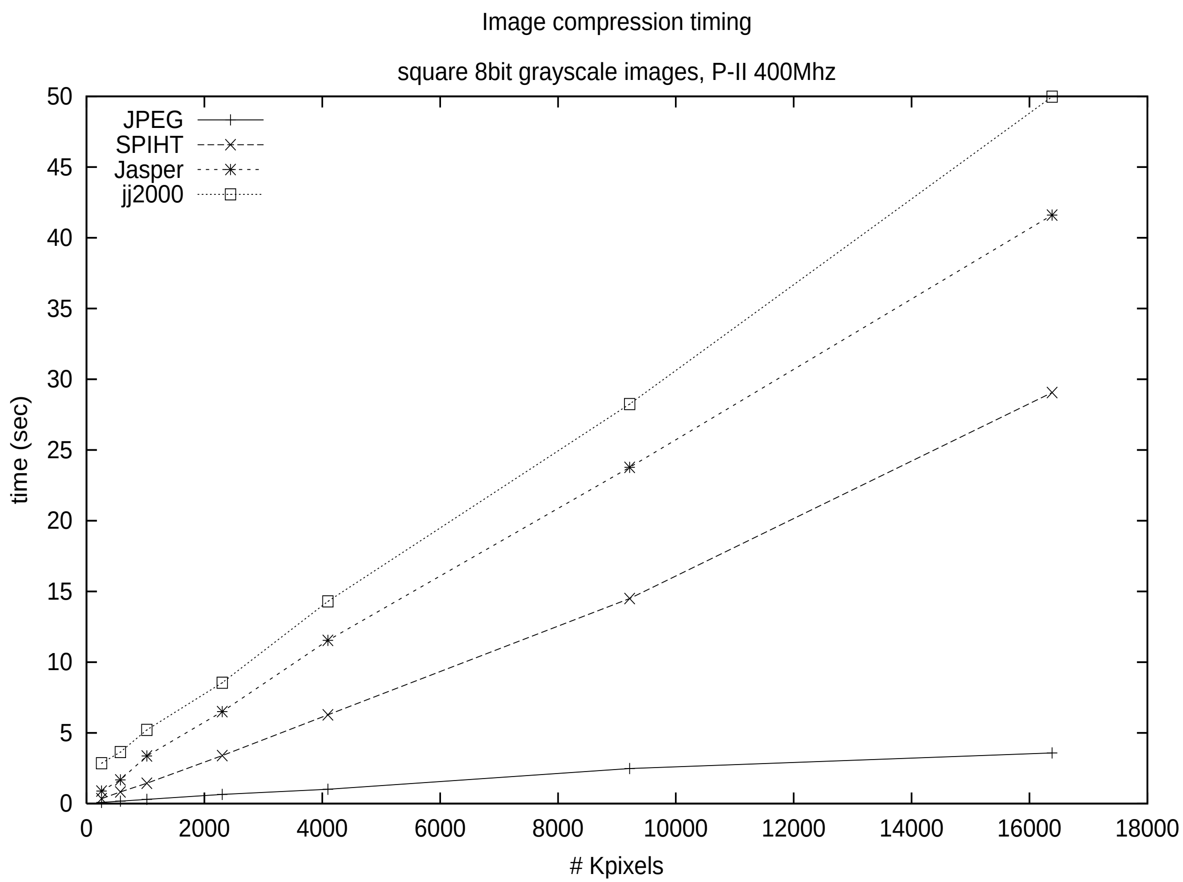
JPEG2000 Parallelization using Java Threads

Peter Meerwald, pmeerw@cosy.sbg.ac.at

Motivation & SMP & JPEG2000 overview
Runtime analysis
Java Multi–threading
Parallelization Idea
Cache Impact
Results & Discussion

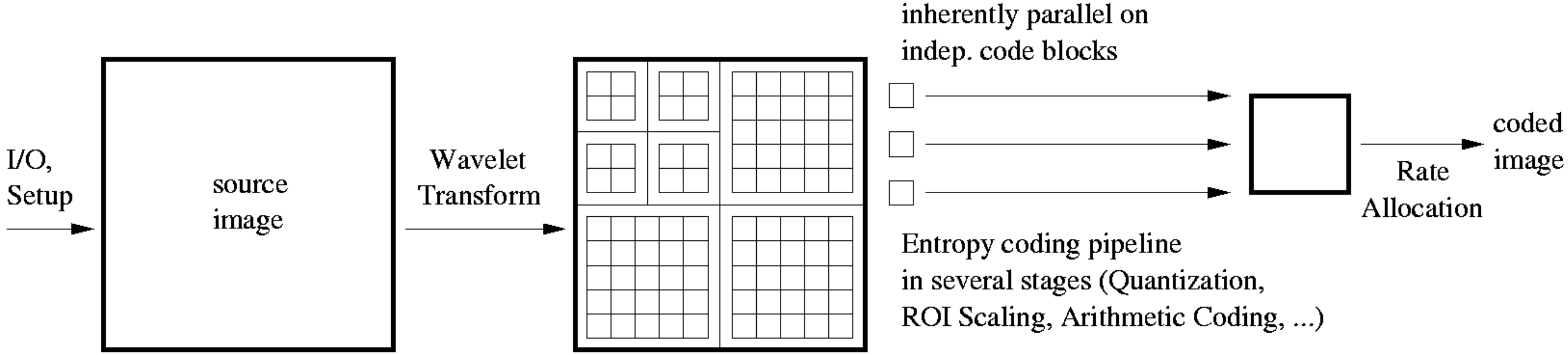
Motivation

- JPEG2000 coding standard
- independent code–blocks
- 'easy' implementation
  http://jj2000.epfl.ch
- Java multi–threading
- runtime requirements
- availability of SMP systems

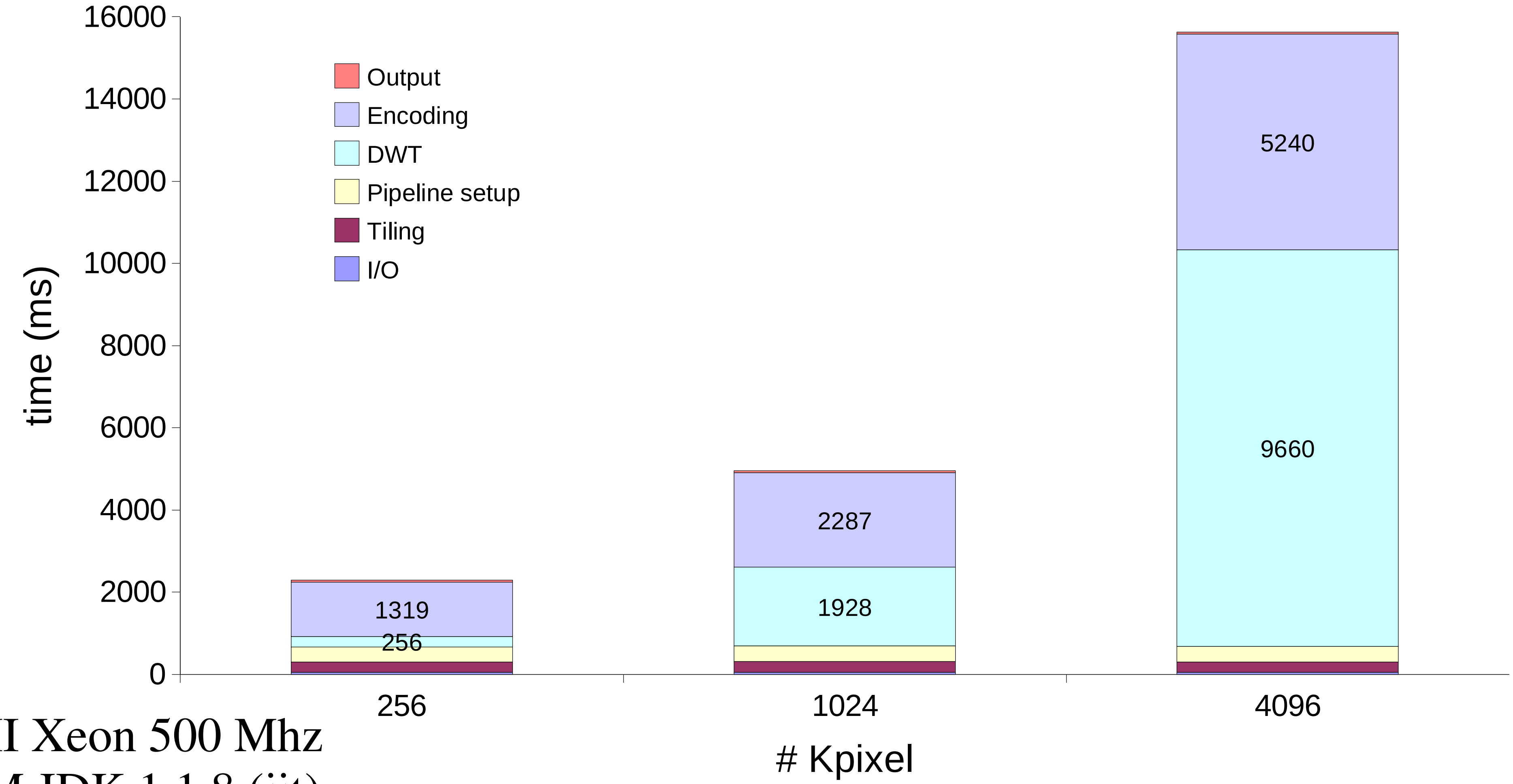


JPEG2000 coding overview

- Wavelet transform
- Entropy coding of independent code−blocks
- Rate Allocation



Runtime Analysis (serial)



P–II Xeon 500 Mhz
IBM JDK 1.1.8 (jit)
jj2000 4.1

Parallelization Idea

Multi–threading in

- Wavelet Transform and
- Encoding stage

- synchronization required between vertical and horizontal filtering
- known workload allows static thread allocation

- independent code–blocks (no synchronization required)
- different runtime for each code–block
- solution: pool of worker threads

Java Multi–threading

Abstract Filtering Thread

```
abstract public class FilterThread extends Thread {
  protected int idx;
  private Barrier barrier;

  public FilterThread(int idx, Barrier barrier) {
    this.idx = idx;
    this.barrier = barrier;
  }

  abstract protected void filterVertical();
  abstract protected void filterHorizontal();

  public void run() {
    filterVertical();
    barrier.block(); // synchronize
    filterHorizontal();
    barrier.block(); // synchronize
  }
}
```

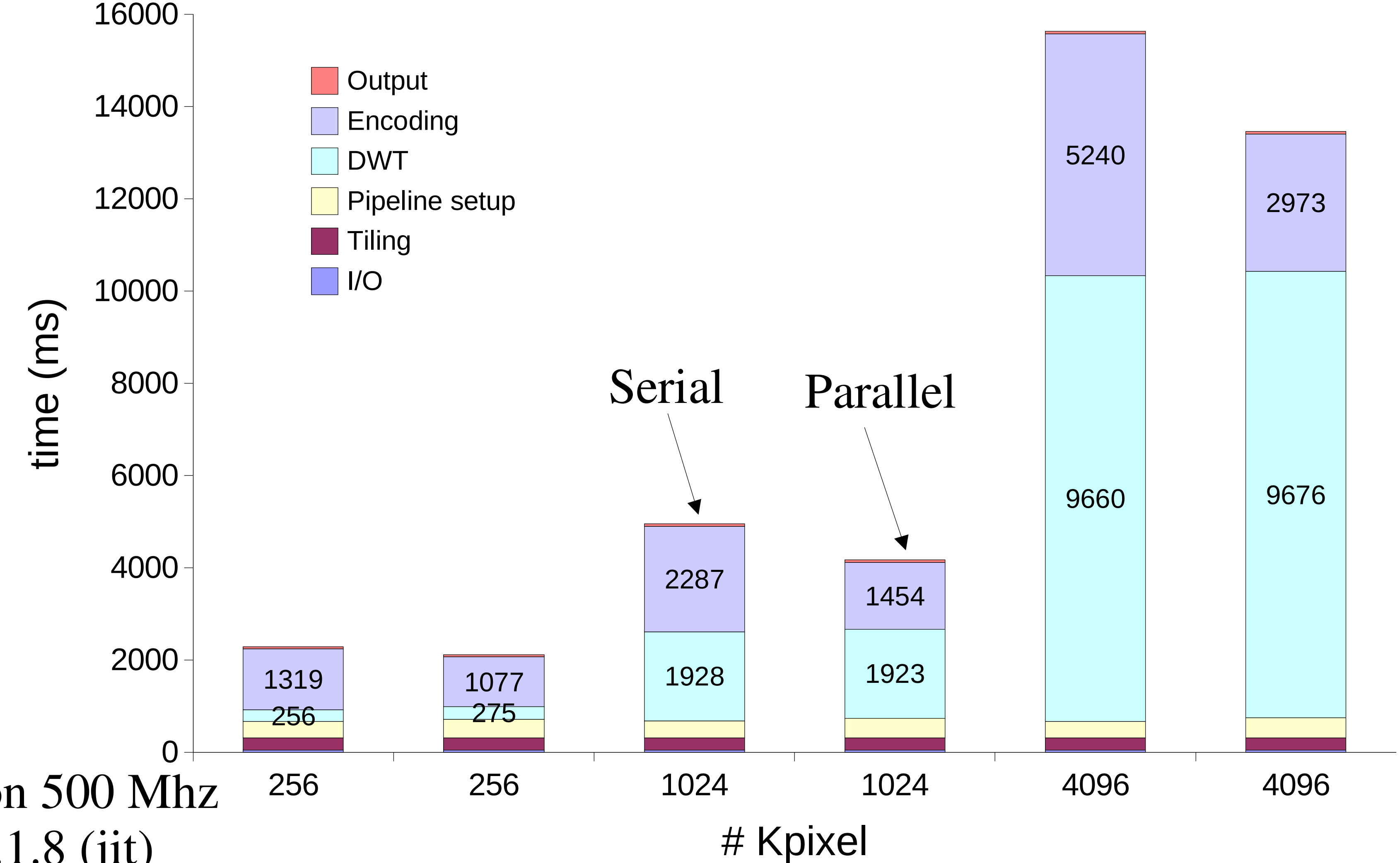
Synchronization

```
public class Barrier {
  private int expected, arrived;

  public Barrier(int expected) {
    this.expected = expected;
    this.arrived = 0;
  }

  public synchronized void block() {
    arrived++;
    if (arrived == expected) {
      arrived = 0;
      notifyAll();  // notify threads to continue
    }
    else{ // need to wait for more threads
      try { wait();} catch (InterruptedException e) { }
    }
  }
}
```

Thread Invokation

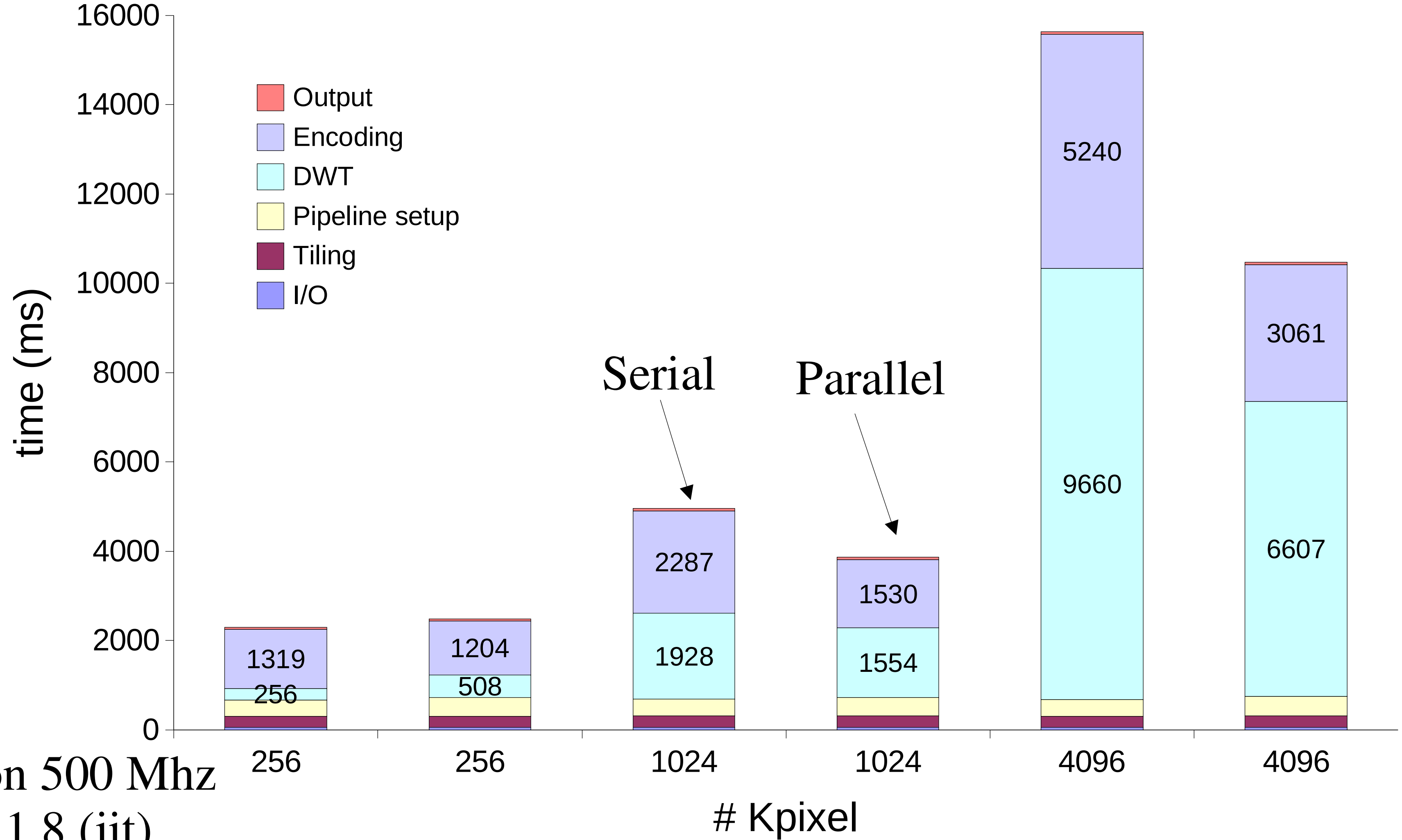
```
private void wavelet2DDecomposition(final DataBlk band, final SubbandAn subband, int c) {
  if (subband.w == 0 || subband.h == 0) return;
  final Barrier barrier = new Barrier(nThreads);

  if (intData) {
    final int[] data = ((DataBlkInt)band).getDataInt(); // global data (for all worker threads)

    class IntegerFilterThread extends FilterThread {
      final int[] tmpVector = new int[java.lang.Math.max(w,h)]; // thread-local data

      public IntegerFilterThread(int idx) {
        super(idx, barrier); // instantiate FilterThread
      }

      protected void filterVertical() { // implemented here (lifting) ... }
      protected void filterHorizontal() { // ... just cut&paste the original serial code, almost same scope }
    }

  for (int i = 0; i < nThreads; i++) {
    threads[i] = new IntegerFilterThread(i); // create threads
    threads[i].start(); // start threads (execute run() method in parallel)
  }

  threads[0].join(); // wait for worker threads
```

Runtime analysis (parallel encoding stage)



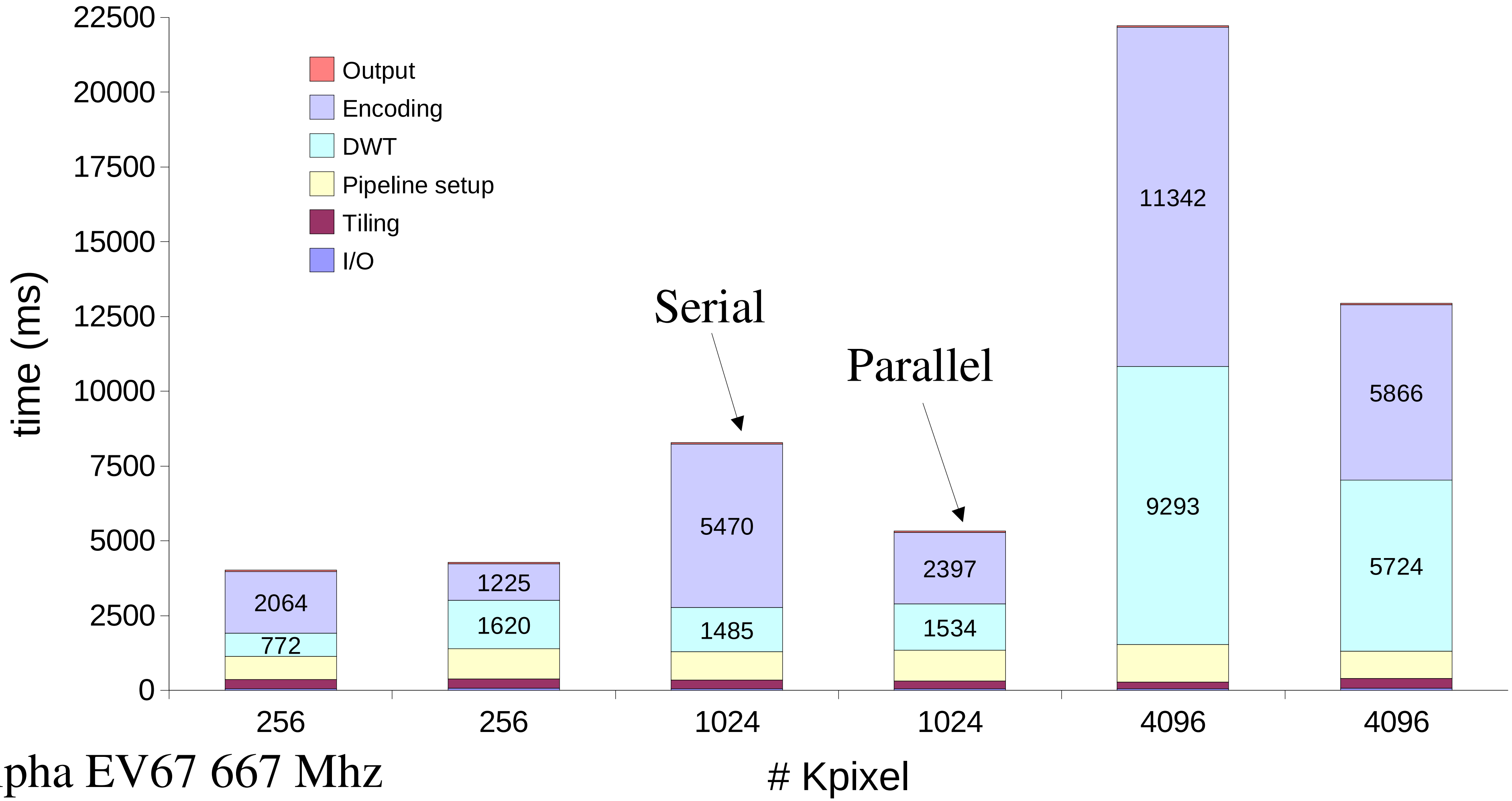
4x P–II Xeon 500 Mhz
IBM JDK 1.1.8 (jit)
jj2000 4.1

Runtime analysis (fully parallel, Intel)



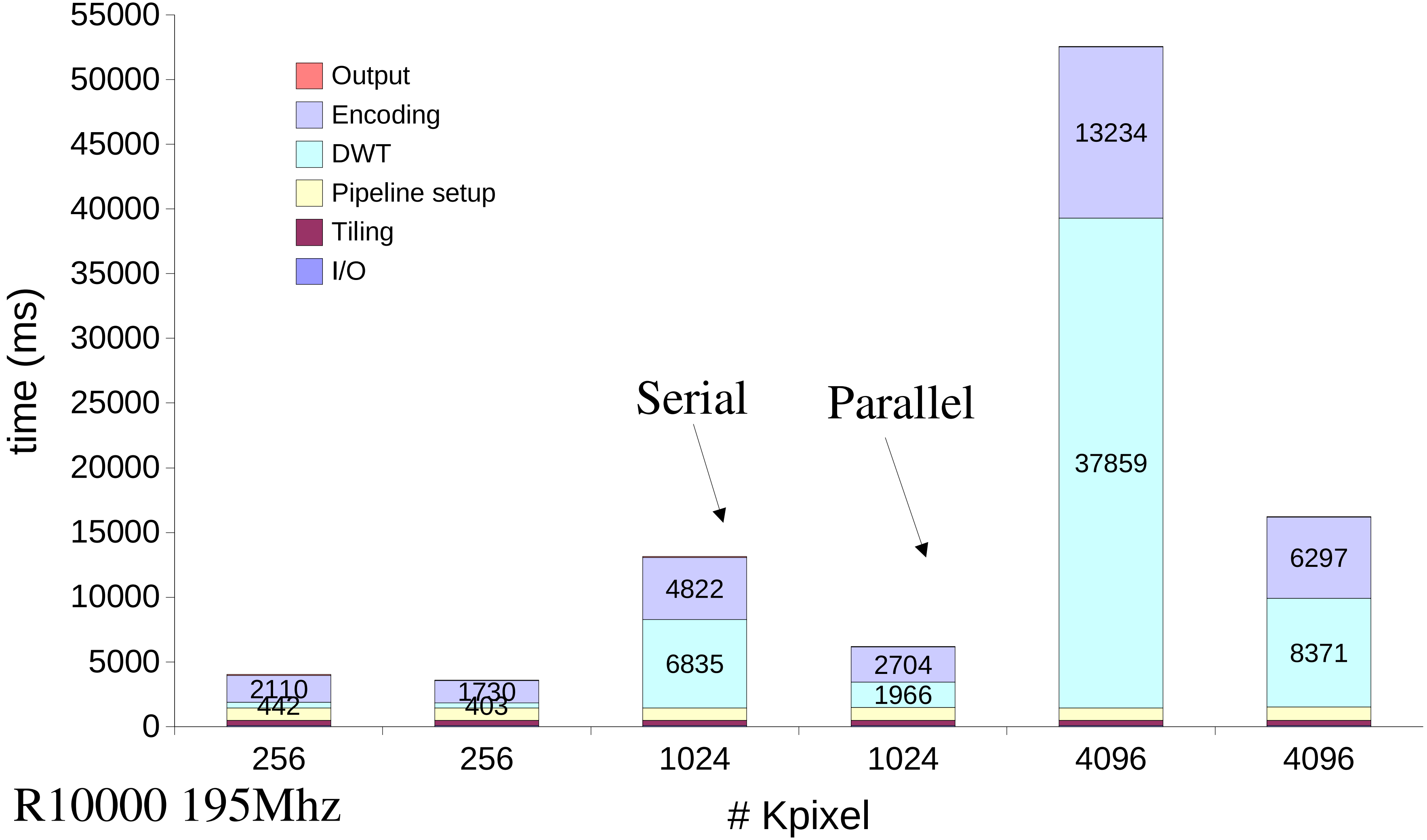
4x P–II Xeon 500 Mhz
IBM JDK 1.1.8 (jit)
jj2000 4.1

Runtime analysis (fully parallel, Alpha)



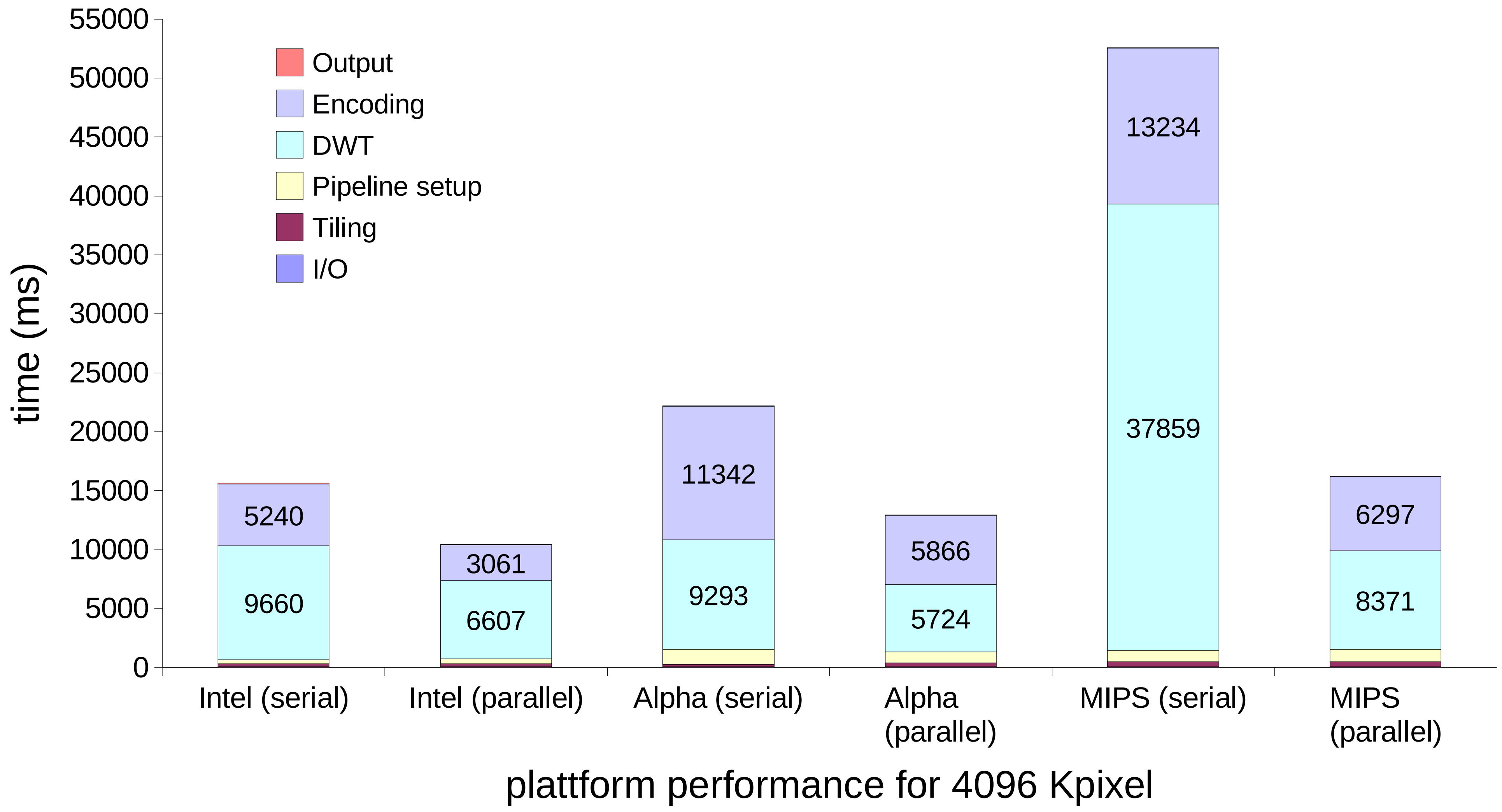
4x Alpha EV67 667 Mhz
Sun JDK 1.22 (jit)
jj2000 4.1

Runtime analysis (fully parallel, MIPS)

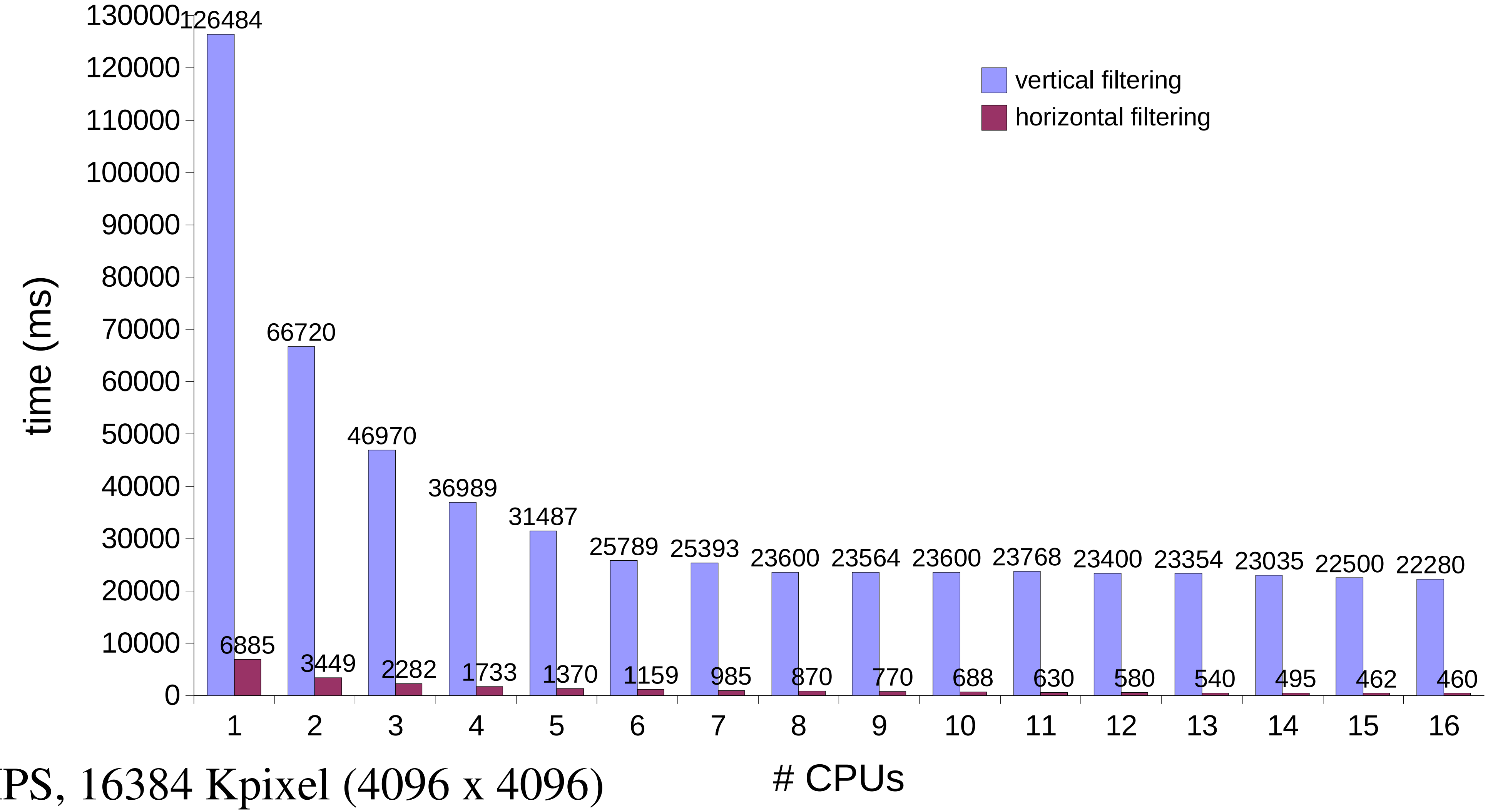


16x MIPS R10000 195Mhz
Sun JDK 1.3 (jit)
jj2000 4.1

Runtime comparison

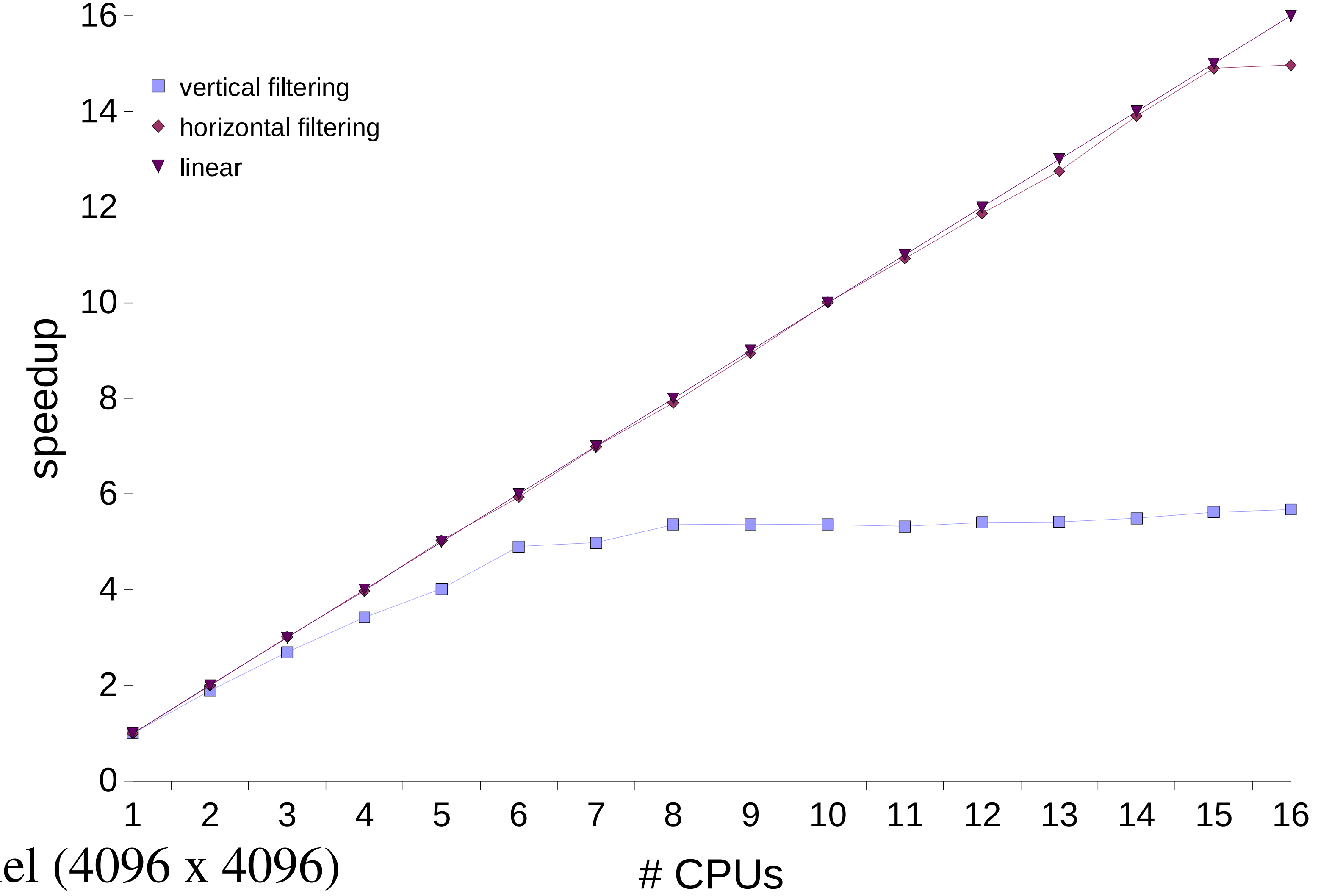


Wavelet Filtering Runtime



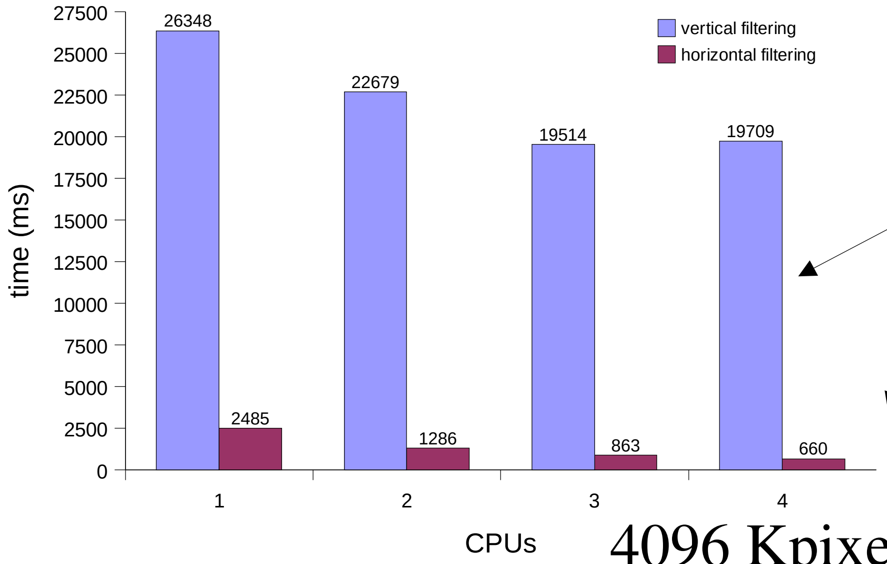
MIPS, 16384 Kpixel (4096 x 4096)
first level decomposition, single chunk

Wavelet Filtering Speedup

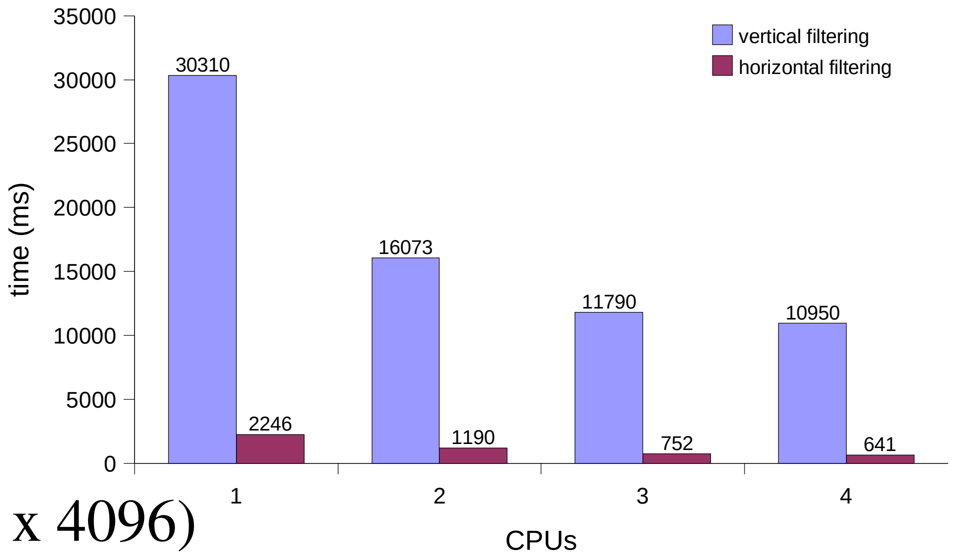


MIPS, 16384 Kpixel (4096 x 4096)
first level decomposition, single chunk

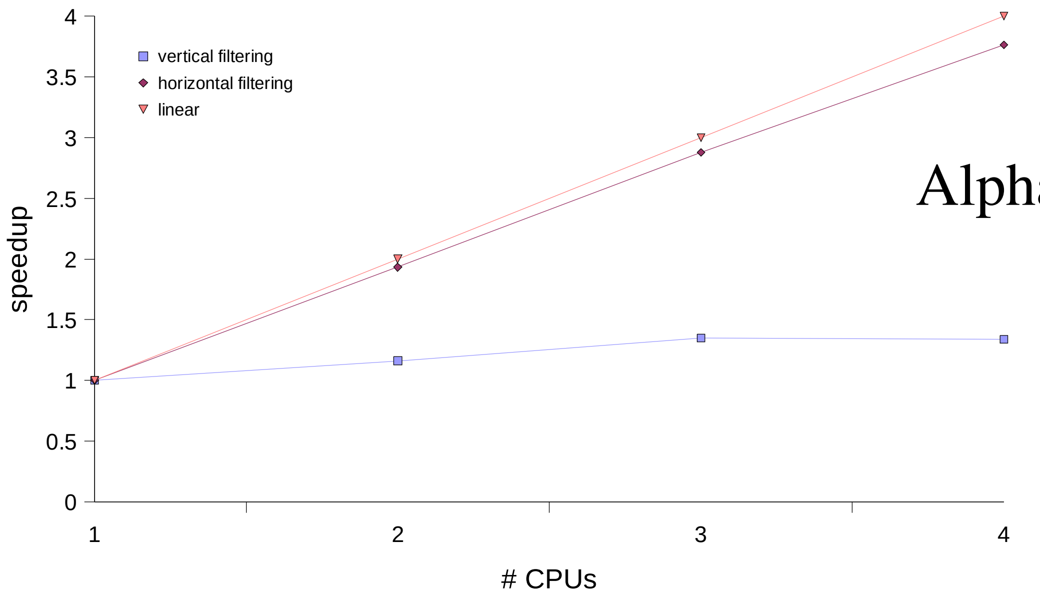
Filtering Results



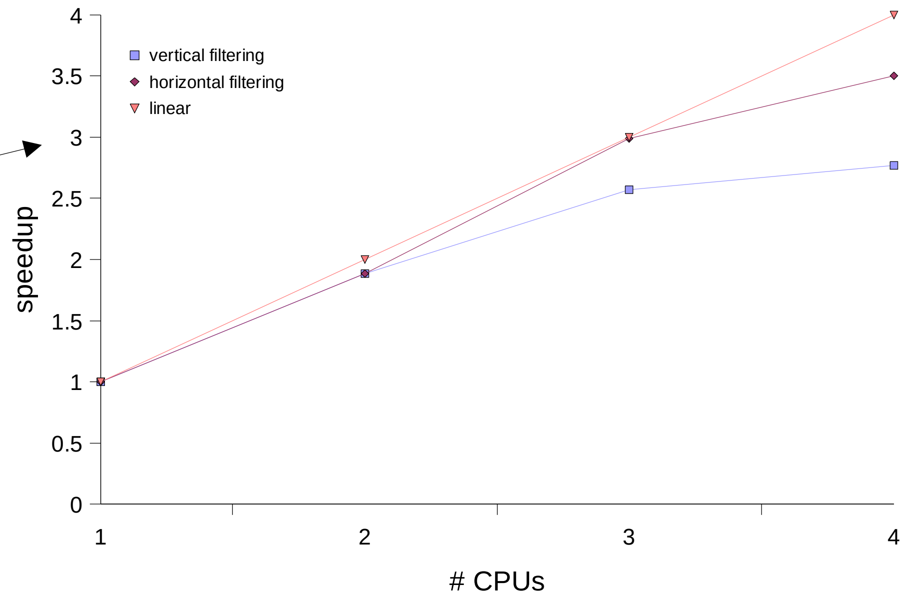
Intel



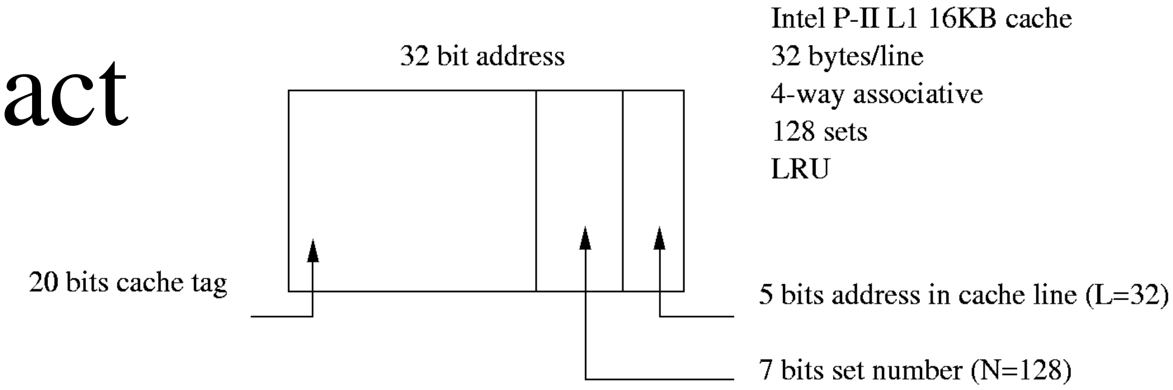
4096 Kpixel (4096 x 4096)
first level decomposition, single chunk



Alpha

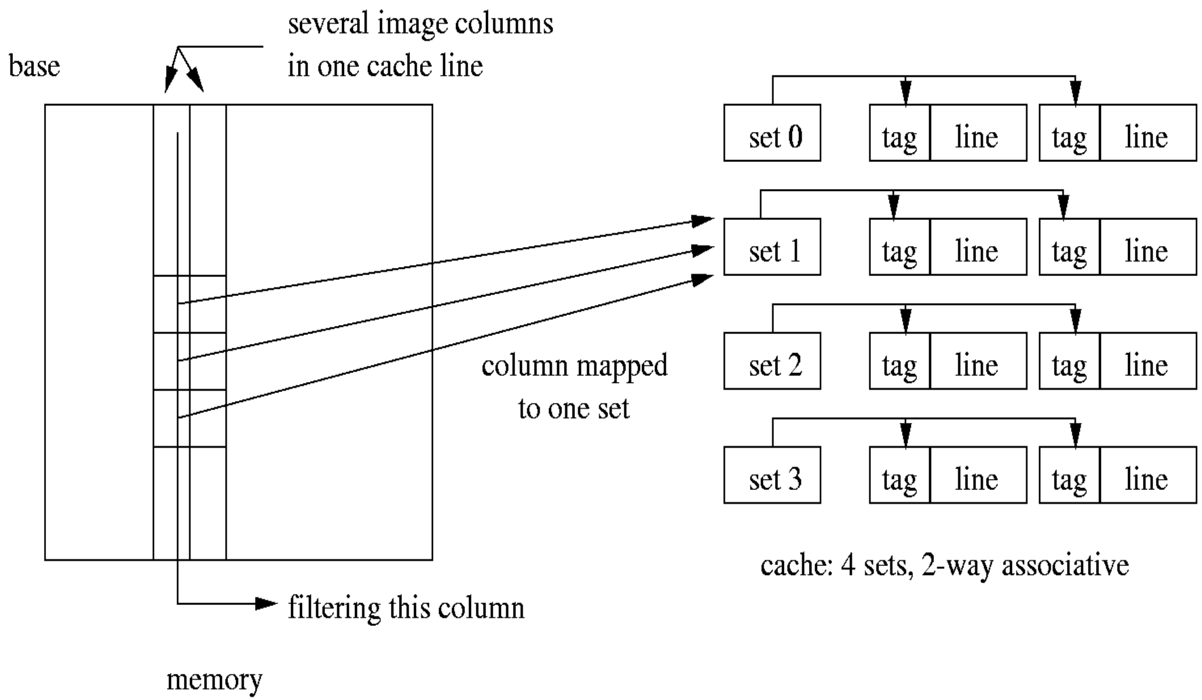


Cache Impact



pathologic memory access pattern for vertical filtering if

- image width is power of 2 and
- image row is larger than 4096 bytes and
- filter length is longer than 4



Improving Cache Hit Rate

1

filter several adjacent columns at the same time

- several columns in one cache line, e.g. L=32 bytes
- only first column access is a miss, subsequent hits
- requires modification in filter code
- would allow SIMD processing: MMX/SSE

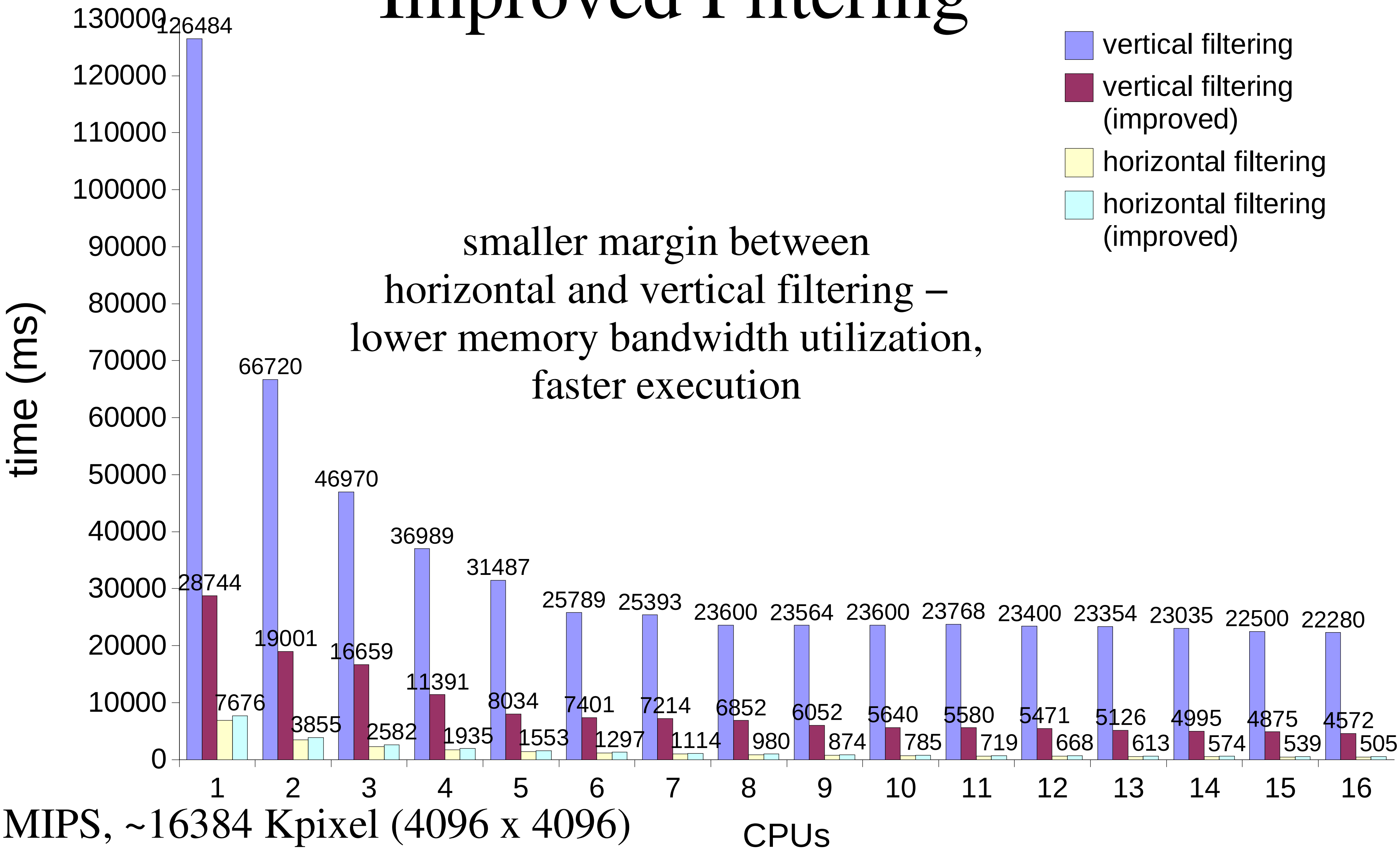
and / or

2

force image width not to be a power of 2

- simple, no modification in filter code
- using more cache sets
- allows cache hits on vertically adjacent pixels

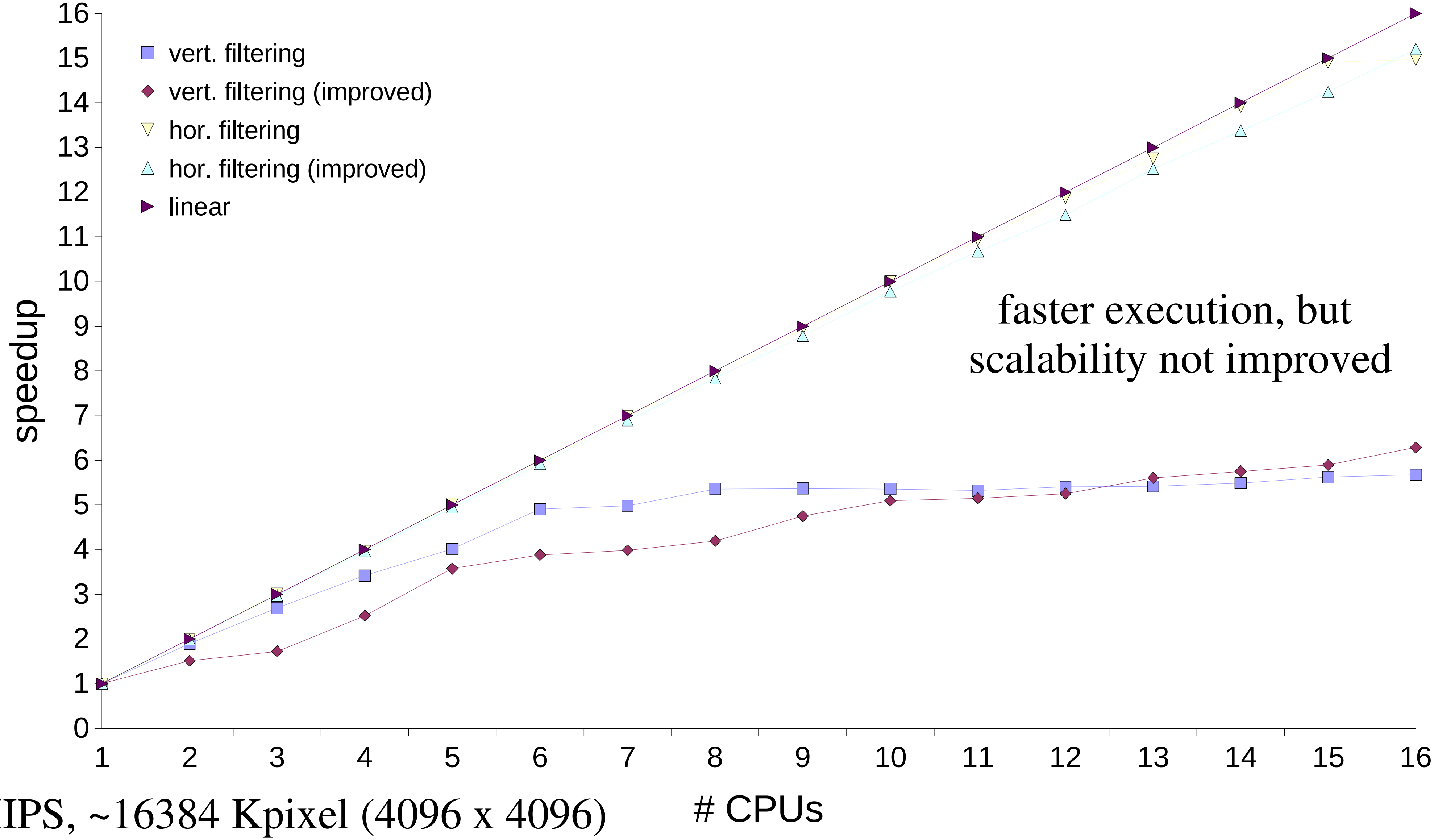
Improved Filtering



smaller margin between
horizontal and vertical filtering –
lower memory bandwidth utilization,
faster execution

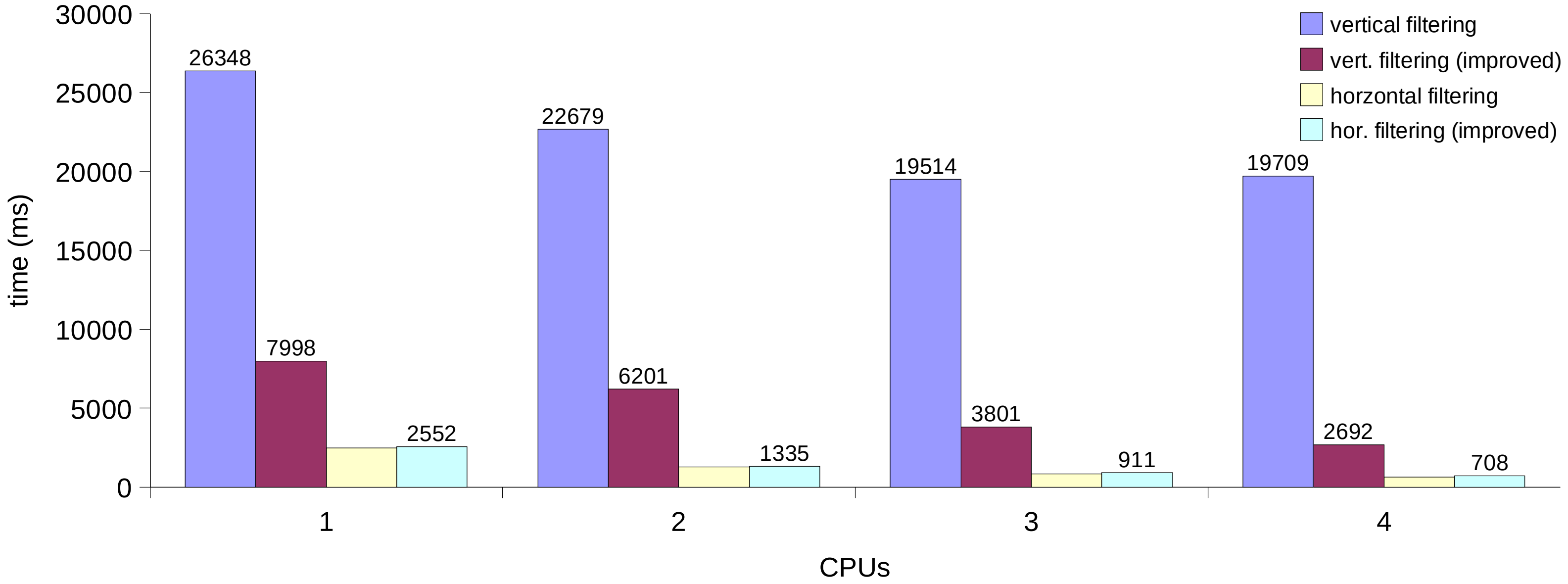
MIPS, ~16384 Kpixel (4096 x 4096)
first level decomposition, single chunk

Improved Filtering Speedup

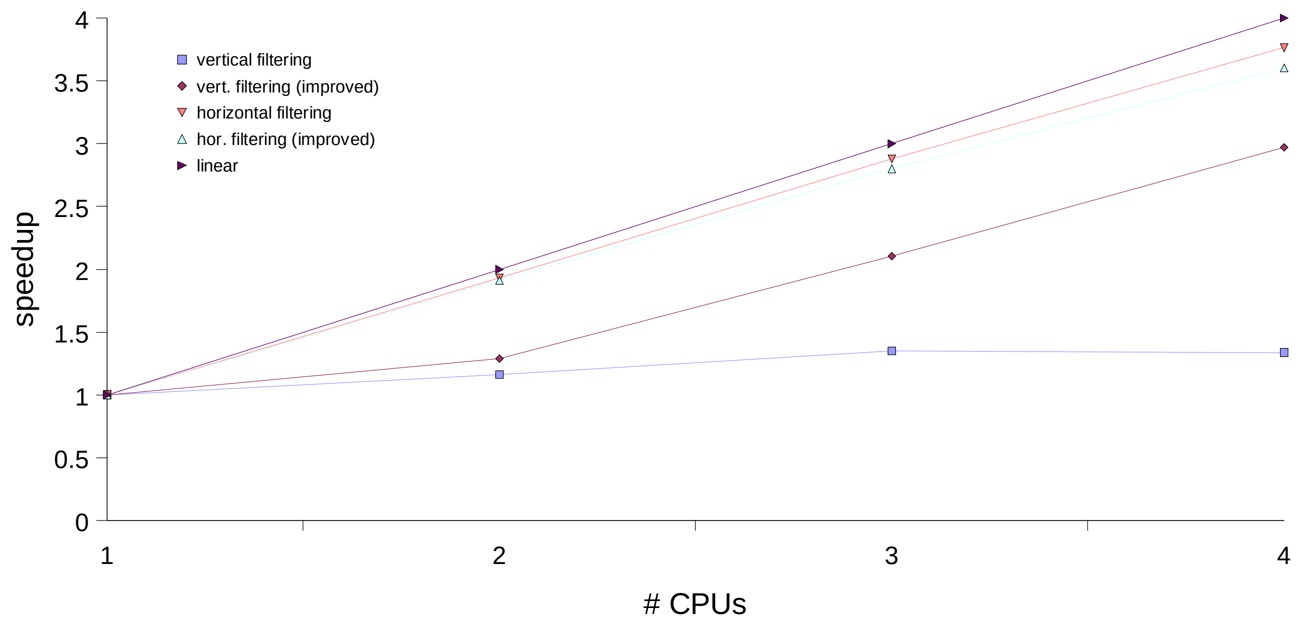


MIPS, ~16384 Kpixel (4096 x 4096)
first level decomposition, single chunk

Improved Filtering (Intel)

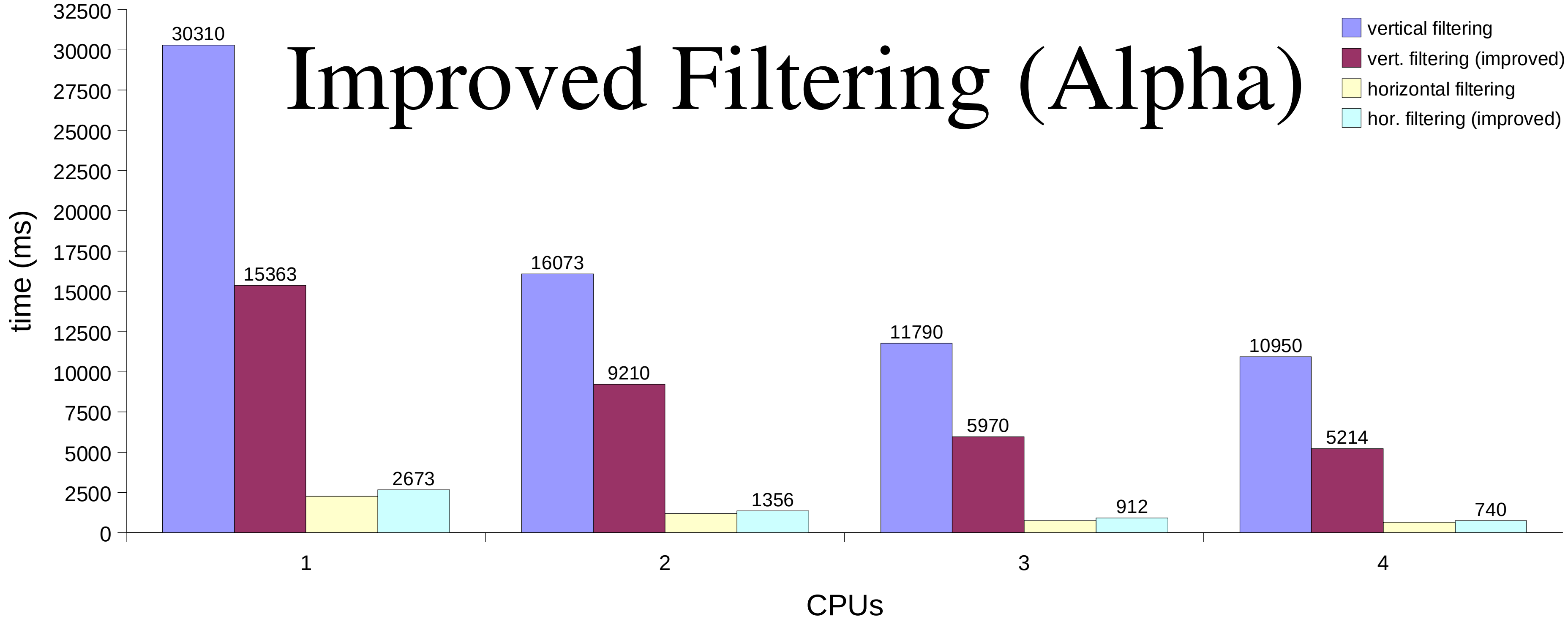


improved runtime,
better scalability

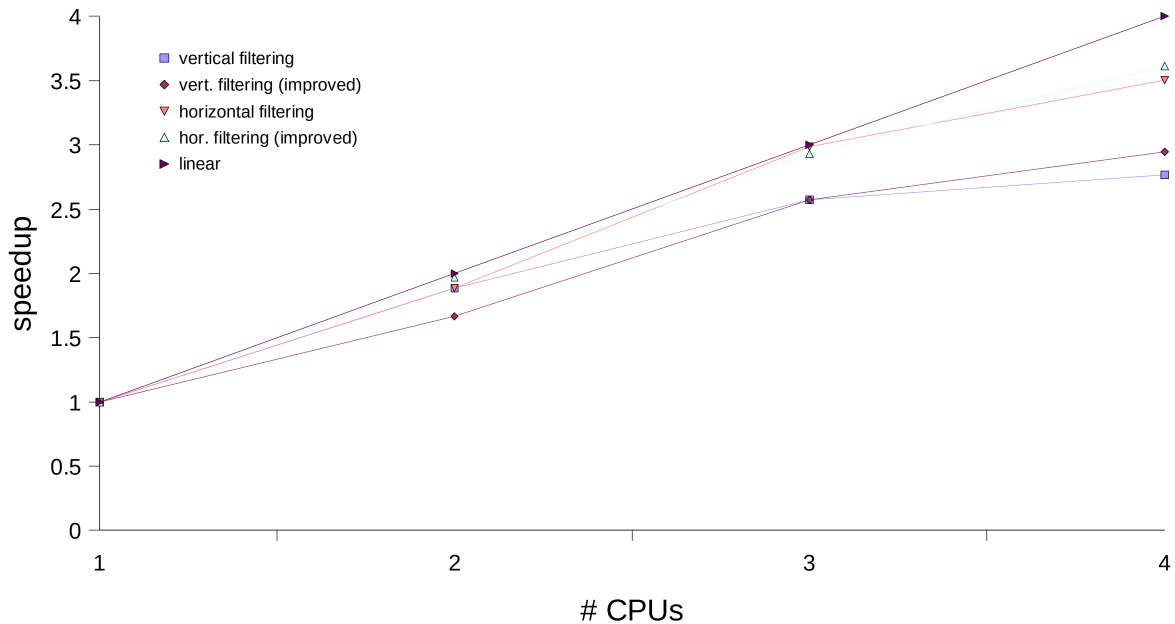


Intel P–II Xeon 500 Mhz
~16384 Kpixel (4096 x 4096)
first level decomposition,
single chunk

Improved Filtering (Alpha)



improved runtime



Alpha EV67 667 Mhz
~16384 Kpixel (4096 x 4096)
first level decomposition,
single chunk

Discussion & Remarks

Speedup
- Intel = 3.37
- Alpha = 1.90
- MIPS = 6.24

Efficiency
- Intel = 0.84
- Alpha = 0.48
- MIPS = 0.39

- simple parallelization using Java
- synchronization still sub–optimal
- further improvements possible (modified filtering)
- filtering as first pipeline stage possible
- discovered cache/memory bottleneck
- best performance using IA–32 architecture