IGP/IWR Summer School

Hardware-aware Scientific Computing

Heidelberg, October 4-15, 2021

IGP = Indo-German Partnership

- A special program financed by DAAD (German Academic Exchange Service) and UGC (University grants commission)
- Funded 2020 - 2024
- Main Partners:
  - Department of Computational and Data Sciences (CDS) at the Indian Institute of Science, Bangalore
  - Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University
- Collaborating Partners:
  - Tata Institute of Fundamental Research
  - Jülich Supercomputing Centre as part of the Forschungszentrum Jülich
  - NVIDIA

Measures

- Mobilities for master students, doctoral candidates, postdocs and professors
  - This might be a possibility for you! Think about it!
- Organization of schools and workshops
  - This is the first such event in a series
- Joint supervision of master and doctoral candidates
- Individual research collaborations
- Explore the possibility of introducing a cotutelle program on the Ph.D. level between IISc and U Heidelberg
- Prepare lectures/materials for a joint course on hardware-aware scalable numerics

Proceeding of the Summer School

- Week October 4-8: Lecture programme
  - https://conan.iwr.uni-heidelberg.de/events/hasc_summerschool2021/
  - Dinner in Heidelberg: Wednesday, 17:00 (german time)
- Week October 11-15: Projects
  - Will be more spontaneous :-)
- Hybrid format
  - For those participating online, please
    - Switch on your microphone and camera if possible
    - Use the chat only if there is no other chance
  - We will leave on the zoom room during the breaks for discussion
- Before asking a question/making a remark for the first time, please introduce yourself shortly

Hardware-aware Scientific Computing

Introduction & Programming Models

Peter Bastian, IGP Summer School, Heidelberg University, October 4-15, 2021

Purpose of the Summer School

- Progress in Scientific Computing happens through
  - Development of new algorithms and theory: reduce flops, solve new problems
  - Increase in compute power
- Increase in compute power is driven by Moore's law, *but*
  - Hardware gets increasingly more difficult to use efficiently
  - Large gap between peak performance and obtained performance
  - This will get worse with the end of Moore's law
  - And by the way: Moore's law will end

➡In this summer school we want to highlight

  - Efficient algorithms *and*
  - Efficient implementations of these algorithms

Contents of this Lecture

- Compact introduction to hardware development
- Overview of programming models with focus on shared memory
- Performance of selected algorithms as examples
- The lecture is highly CPU centric

Hardware

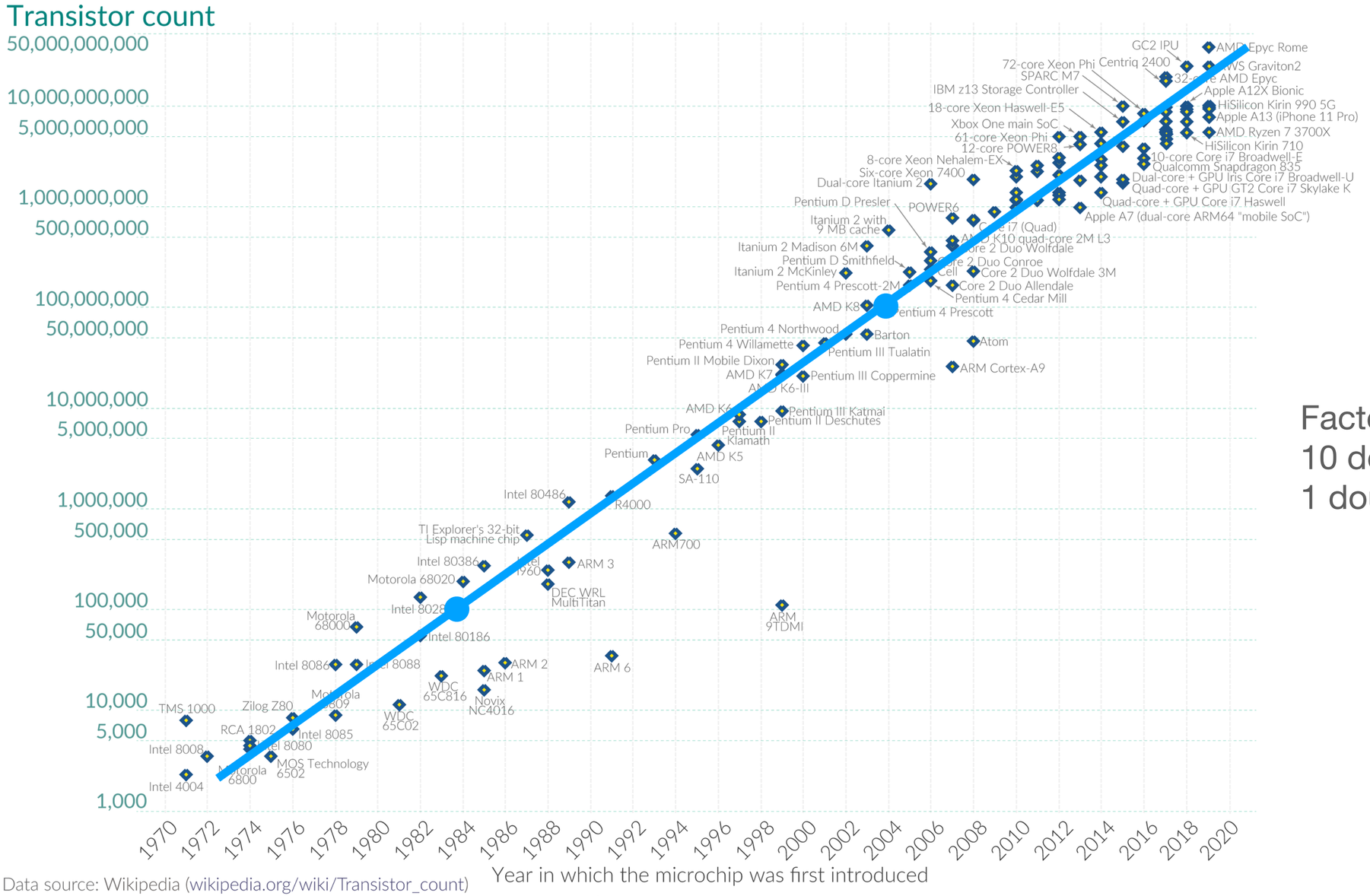
Moore’s Law and Dennard Scaling

- Exponential growth of computing power of digital computers in the last 75 years changed the life of nearly everybody!
- *Moore’s law*: *Number of transistors put economically on a chip doubles every two years (Gordon Moore 1965, changed from 1 year to two years in 1975)*
- *Dennard Scaling: Power density (W/m^2) can be kept constant at scaling even with increasing clock rate by lowering supply voltage*
- Dennard scaling ended 2004
- Moore’s law has significantly slowed down in recent years
- Computer *performance* is a consequence of
  1) Moore’s law + Dennard scaling
  2) Improvements in computer architecture (what to do with all the transistors?)
  3) Improvement in computational algorithms

Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Factor 1000 in 20 years,
10 doublings in 20 years
1 doubling every 2 years

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

Very popular minicomputer by Digital Equipment Corporation

Increase of **single chip performance** on SPEC integer benchmark
(at most 4 cores per chip)

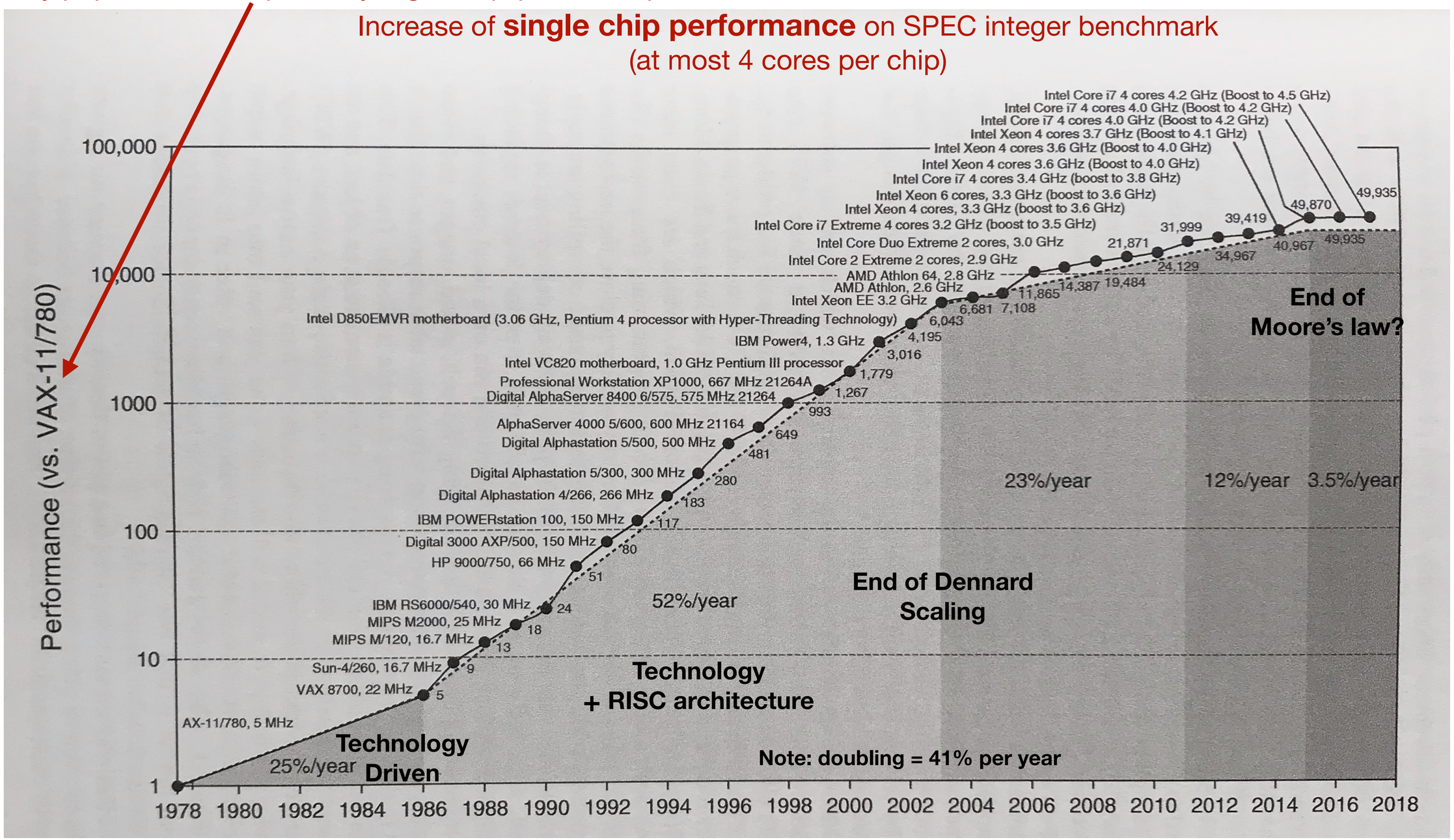


Image credit: Hennesy/Patterson. Computer Architecture

End of Dennard Scaling in Detail

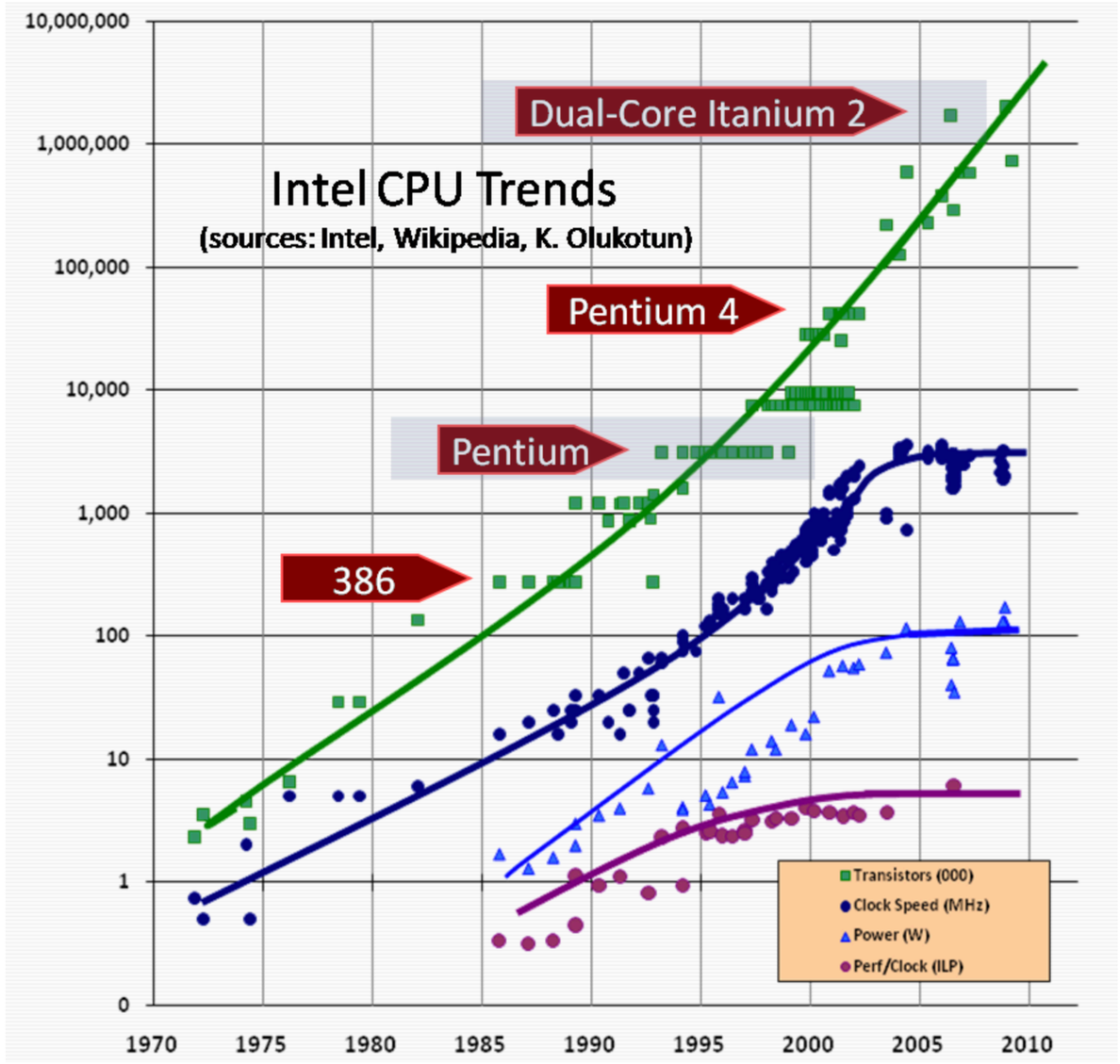
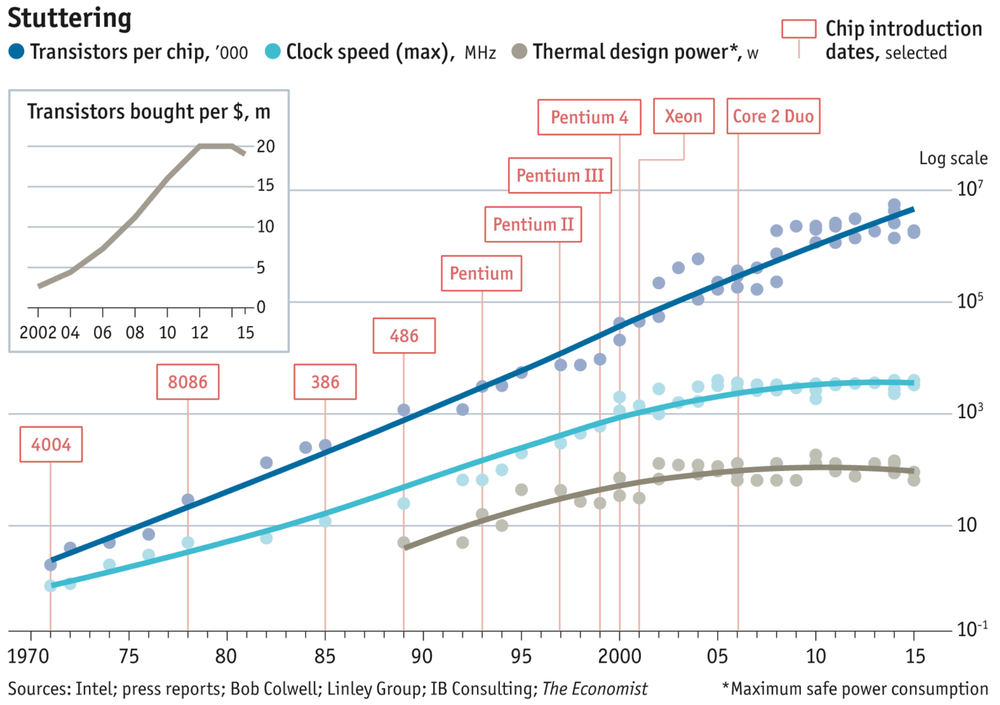


Image credit: Herb Sutter. *The Free Lunch Is Over.* Dr. Dobb's Journal. 2005.

Dramatic events in 2003:

- End of Dennard scaling lead to stagnation of clock rate
- Improvement of instruction level parallelism (ILP) came to halt: This ended the automatic increase of instructions executed per clock
- Way out: **Multicore architecture**



Levels of Parallelism (What to do with silicon)

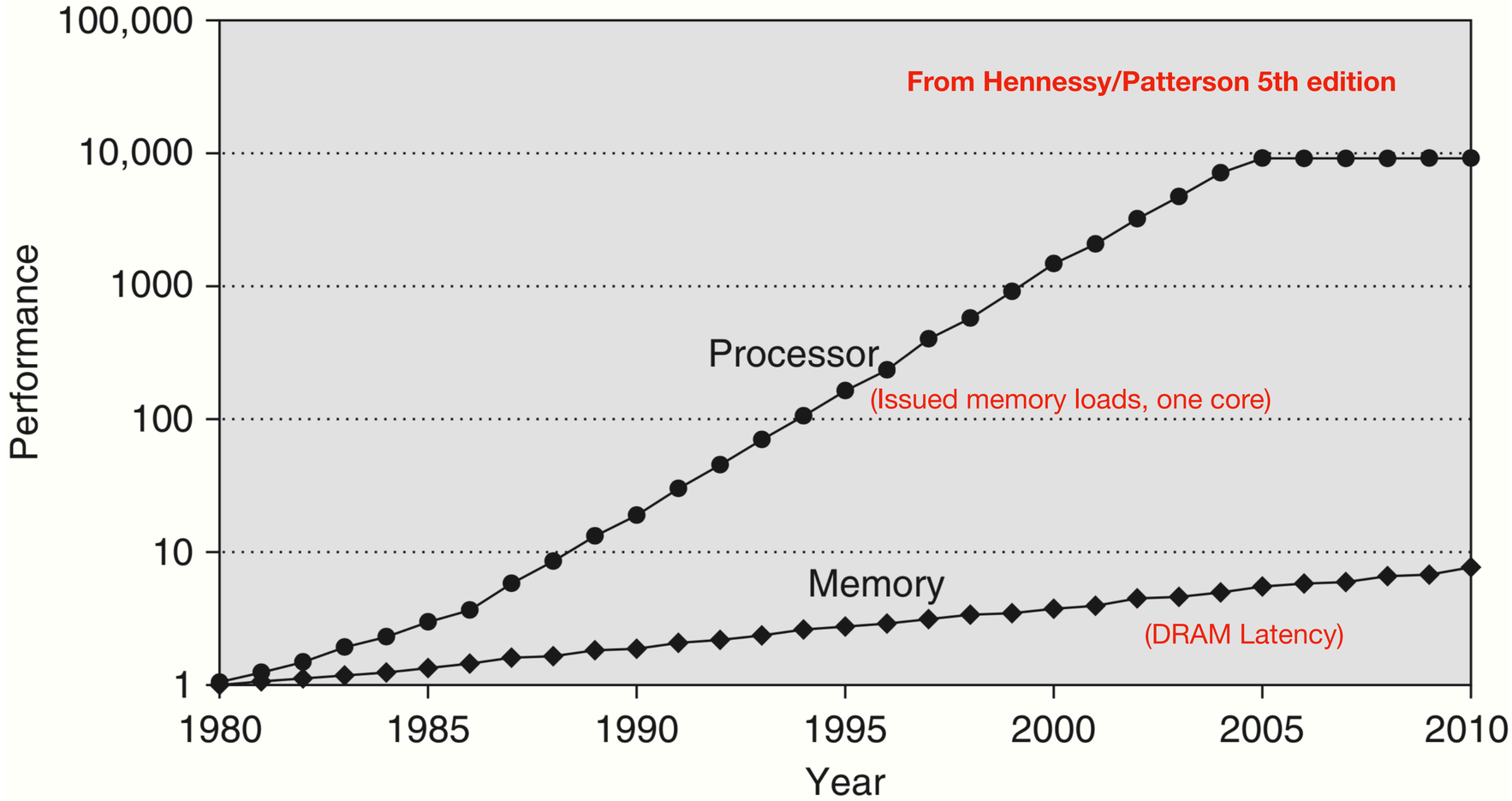
- *Bit-level parallelism (BLP)*
  - 1,4,8,12,16,32,64 Bit processors (width of Registers, address and data wires)
- *Instruction level parallelism (ILP)*
  - Pipelining (assembly line, time parallel execution) of instructions
  - Superscalar execution: >1 instruction/clock
  - Enabled by RISC (reduced instruction set computer)
  - Part of RISC: Load/Store architecture
- *Data level parallelism (DLP)*
  - Vector/Matrix instructions as special form of superscalar execution
  - Also pipelined (many of them executed in parallel and overlapping)
- *Thread level parallelism (TLP)*
  - Independent instruction streams with shared memory access
- *Message level parallelism (MLP, request level parallelism in Hennessy/Patterson)*
  - Independent instructions with private memory and message passing

SIMD Instructions in Microprocessors

- Combines ILP and DLP
- Used in CPUs and GPUs today
- Peak performance = SIMD instructions!
- Introduced in Intel processors ~20 years ago
- Operate on 16 (32 in AVX512) SIMD registers
- Support various integer, SP and DP ops

Name	Year	Width (Bits)	Doubles
SSE	1999	128	No
SSE2	2001	128	2
AVX	2010	256	4
AVX2	2013	256	4 (fma)
AVX512	2017	512	8 (fma)

Memory



- Memory performance grew slower than processor performance for decades
- Memory bandwidth grows a bit faster than latency (not shown)
- Resulted in „memory gap“
- Remedy is the „memory hierarchy“

Exploring the Memory Hierarchy: Pointer Chasing

- From Hennesy/Patterson, 6th ed., Figure 2.32 on page 151
- Chose an array x of integers with length n and a stride $s < n$ with s dividing n
- Then set $x[i] = (i + s) \bmod n$ for $0 \leq i = ks < n$
- „Pointer-chasing“ or „Index-chasing“ then means to execute the loop

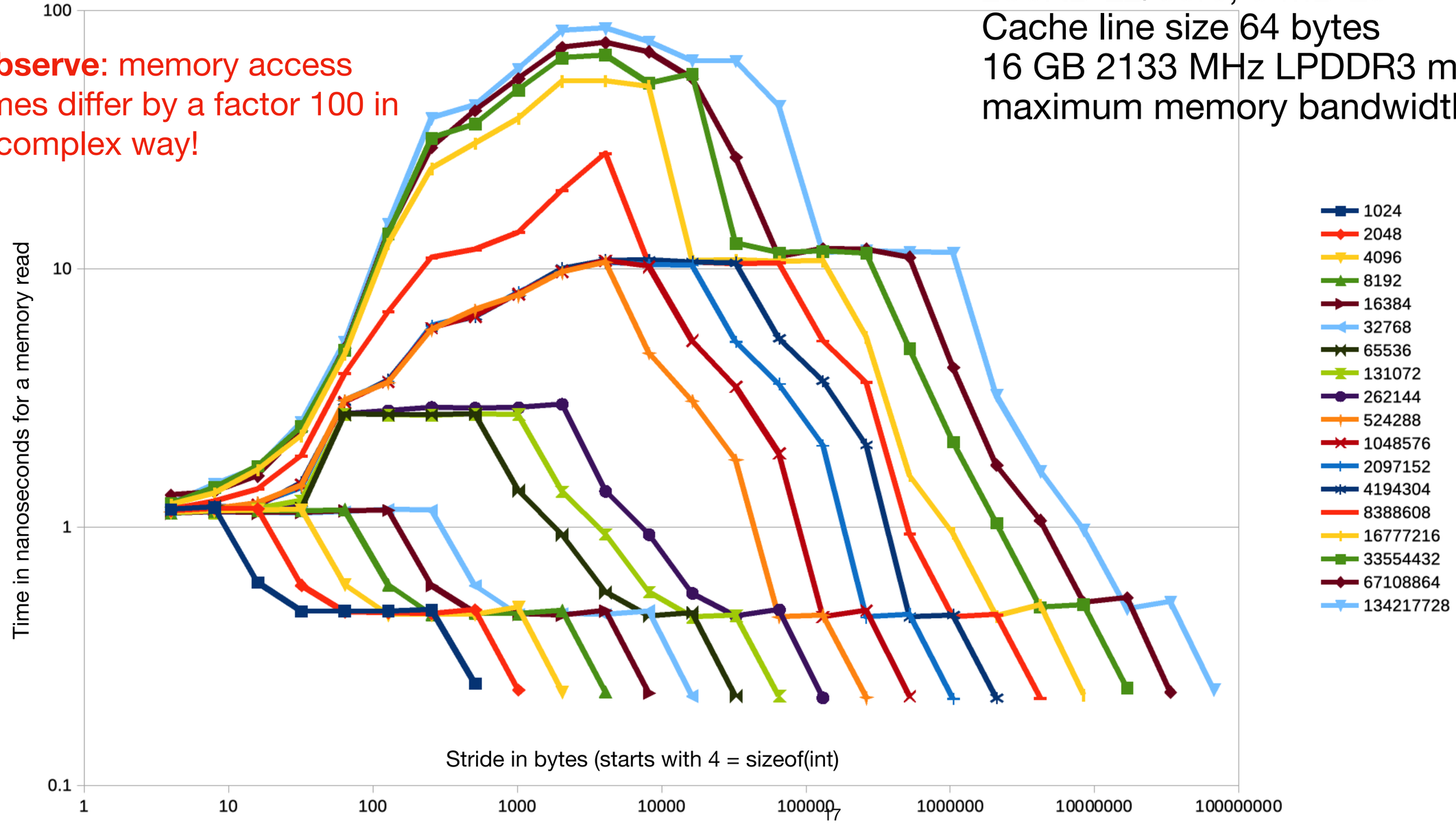
```
i=x[0];
while (i!=0) i=x[i];
```

- This means we do memory reads every s'th integer from the array
- In total there are n/s reads and we repeat s times; so an „experiment“ does n reads, irrespective of s
- Now equip this with some reliable timing

Pointer Chasing: Results

Intel(R) Core(TM) i7-8559U CPU @ 2.70GHz,
4 cores, 64KB L1/core (32I,32D),
256KB L2/core, 8 MB L3
Cache line size 64 bytes
16 GB 2133 MHz LPDDR3 memory
maximum memory bandwidth: 37.5 GB/s

Observe: memory access times differ by a factor 100 in a complex way!



Interpretation of the results

- $s = 1$. Independent of n we observe an execution time of about 1.2 ns, corresponding to 5 or 6 clock cycles (4,5 GHz turbo mode)
  - The chasing contains a data dependence resulting in a pipeline stall that limits execution time
  - The required memory bandwidth 4Byte/1.2 ns = 3.33 GB/s is easily delivered by main memory, so the time is independent of n!
- $s = n/2$. These are the bottom data points at 0.22 ns = 1/4.5 GHz.
  - Only two memory locations are read, so memory is not an issue here.
  - Only one comparison is done; Branch prediction and speculative execution achieve full pipelining without any stalls
- The plateaus correspond to the cache levels; in particular consider
- Fixed n.
  - For small strides $s \leq 8$ (2 ints per cache line are read) memory bandwidth is sufficient and instruction exec with pipeline stalls is the limiting factor
  - For stride $s \geq 16$ (1 int per cache line is read) memory bandwidth becomes the limiting factor and we see the transfer rates of the memory
  - For s reaching the range of n. Only n/s cache lines are needed and we get back down the memory hierarchy again

STREAM Benchmark

- The STREAM benchmark [1] is a very well known benchmark to measure memory bandwidth
- Invented by „Dr. Bandwidth“ John D. McAlpin in 1995
- It times four operations on double precision float vectors:

  copy: x = y
  scale: x = sy
  add: x = y + z
  triad: x = y + sz

[1] McCalpin, John D., 1995: “Memory Bandwidth and Machine Balance in Current High Performance Computers”, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.

STREAM Results

- On my notebook compiled with gcc10 and -Ofast -fargument-noalias -march=native -fopenmp

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           28745.3     0.007587     0.005566     0.016805
Scale:          18995.9     0.009139     0.008423     0.010568
Add:            19367.3     0.013923     0.012392     0.015311
Triad:          17996.5     0.015054     0.013336     0.020023
```

- Nominal memory bandwidth is 37.5 GB/s, so we are quite far away.
- The problems are explained in Georg Hager's blog https://blogs.fau.de/hager/archives/8263:
  - Copy is translated to a memcpy call, that's why it is faster
  - With option -ffreestanding we get

```
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           15696.9     0.010748     0.010193     0.011705
Scale:          15626.3     0.010790     0.010239     0.011965
Add:            17822.5     0.014725     0.013466     0.016031
Triad:          17003.1     0.017150     0.014115     0.026462
```

- The problem comes from the accounting. With a write-back cache, the cold write to the result x actually needs an additional read before, which is not accounted for by the code
- The results need to be multiplied with 3/2 for copy/scale and 4/3 for add/triad
- Setting OMP_NUM_THREADS to a lower number might also help

Implications for Software (Roofline Analysis)

Machine Intensity

- Assume a *hypothetical* algorithm using all available machine resources:
  - It performs at peak floating point performance P_m in Gflops/sec
  - It uses the maximum memory bandwidth M_m in Gbytes/sec (the most relevant bandwidth. Might be main memory or cache level x)
- Define the *machine intensity* $I_m = \frac{P_m}{M_m}$ with unit flops/byte
- Our hypothetical algorithm performs I_m flops for each byte transfered to/from memory
- The reciprocal $B_m = 1/I_m$ in bytes/flop is called *machine balance*
- Example i7-8559U CPU @ 2.70GHz: $P_m = 180$ Gflops/sec, $M_m = 37$Gbytes/s, so $I_m = 4.86$ flops/byte

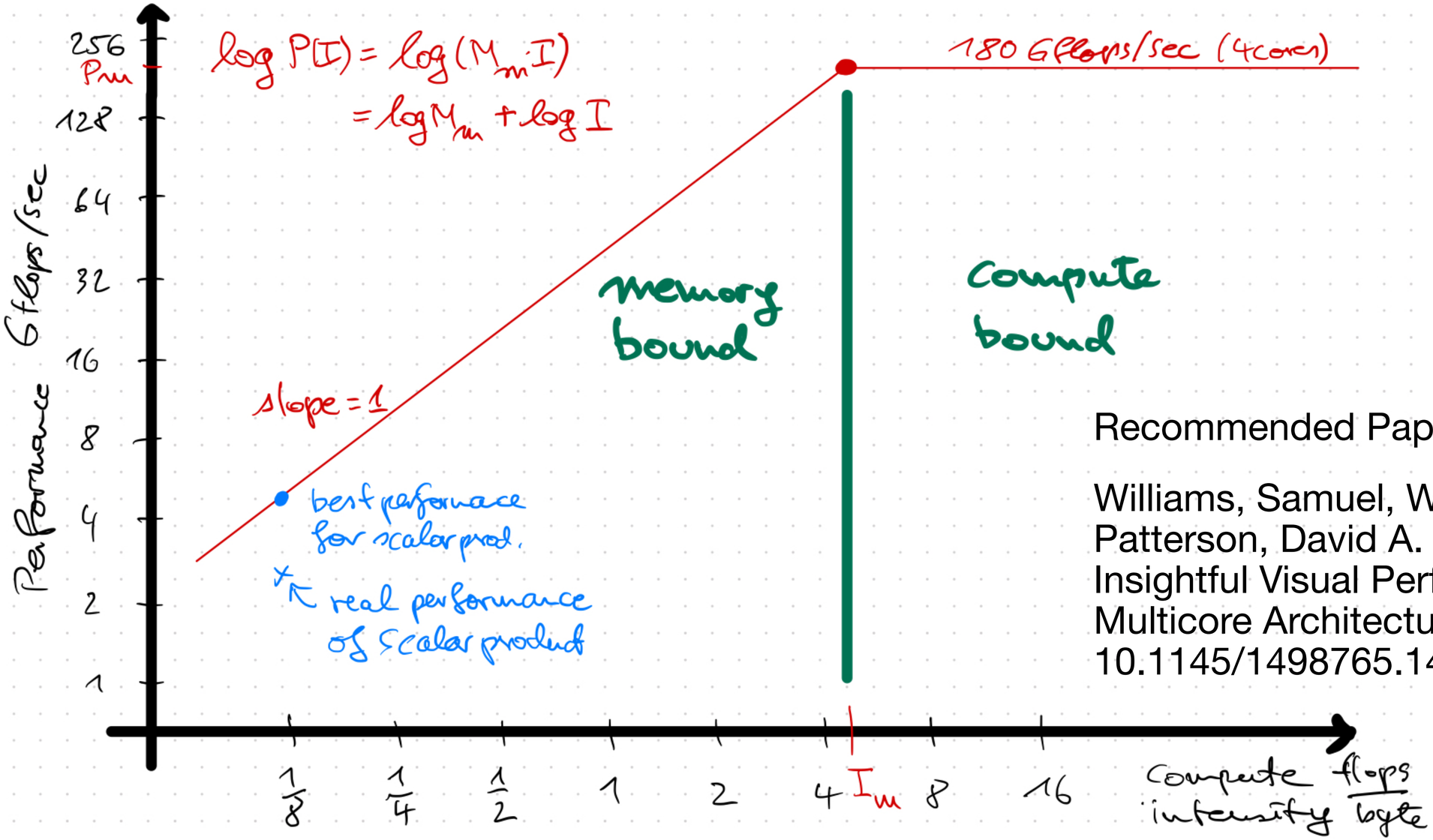
Roofline Analysis

- Now consider a specific algorithm for solving a problem and its implementation
- The idea is to *model* the performance of the implementation on a given machine *as a function of its intensity and machine characteristics*

$$P(I) = \min(P_m, I \cdot M_m)$$

- Fundamental classification of algorithms on a *given* machine:
  Memory-bound algorithms: $I < I_m$, performance limited by M_m
  Compute-bound algorithm: $I > I_m$, performance limited by P_m
- *Example*: Scalar product. $I = 2$ flops/16 bytes = 1/8 flops/byte. With 40 Gbytes/sec memory bandwidth, the algorithm will perform at $P = I \cdot M_m = 0.125 \cdot 40 = 5$ Gflops/sec (if $P \leq P_m$).
- **Only compute bound algorithms may reach peak performance**
- Note: In general performance is less than the ideal performance due to various factors, e.g. pipeline stalls, loop overhead, cache misses that were not anticipated, TLB miss, ..

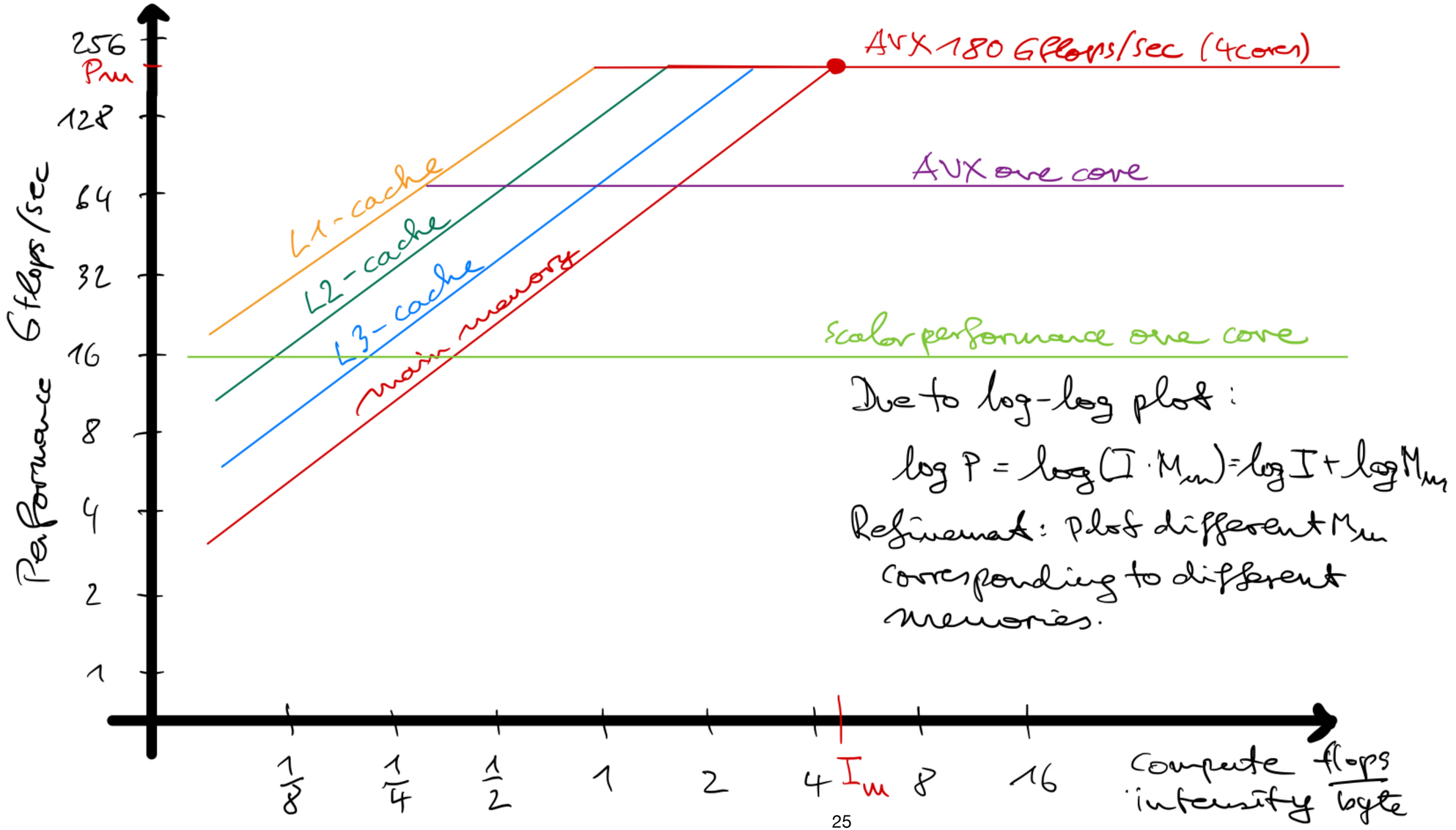
Roofline Diagram



Recommended Paper:

Williams, Samuel, Waterman, Andrew, & Patterson, David A. (2009). Roofline: An Insightful Visual Performance Model for Multicore Architectures. http://doi.org/10.1145/1498765.1498785

Roofline Diagram Refined



Recipe for Performance Optimization

1. Determine compute intensity I
2. If ($I < I_m$) : try to improve compute intensity
   - Optimize access patterns (i.e. *reduce memory bandwidth* to the relevant level of the memory hierarchy
     - Blocking (tiling)
     - Loop fusion
     - Array of structures (AoS) vs. Structure of Arrays (SoA)
3. If ($I \geq I_m$) : optimize instruction rate (i.e. improve pipelining)
   - SIMD
   - Reduce loop overhead (loop unrolling, loop unrolling + vectorization = strip mining)
   - Instruction reordering (avoid pipeline stalls)
4. Use a different algorithm? *But*:
   - Of course without superfluous flops!
   - Sometimes algorithms with worse complexity have higher compute intensity. Is the run-time of the worse algorithm really better?

BLAS

- BLAS = Basic Linear Algebra Subroutines. Library for linear algebra kernels introduced in 1979
  - Level 1: vector ops, e.g. *daxpy* $y = \alpha x + y$, $O(n)$ operations on $O(n)$ data
  - Level 2: matrix-vector ops, e.g. $y = \alpha Ax + \beta y$, $O(n^2)$ operations on $O(n^2)$ data
  - Level 3: matrix-matrix ops, e.g. $C = \alpha AB + \beta C$, $O(n^3)$ operations on $O(n^2)$ data
- Compute intensity on level 1, 2 is constant, algorithm would typically be memory bound, unless the constant is large enough
- Compute intensity on level 3 is $O(n)$, so for n large enough there is a chance to make the algorithm compute bound
  - Peak FP performance might still be a challenge: SIMD, register pressure, loop overhead, etc., see matmul example

Roofline for Matrix Multiplication

- Want $C = C + AB, A, B, C$ square matrices of size $n = MN$
- Write algorithm in block form: $C_{ij} = C_{ij} + \sum_{k=1}^{N} A_{ik} B_{kj}$
- To compute product $A_{ik} B_{kj}$ of $M \times M$ block matrices:
  - Flops made: $2M^3$
  - Bytes loaded: $2 \cdot 8M^2$. Assumes we load A and B but have full reuse on C
- Intensity then is $I = \frac{2M^3}{16M^2} = \frac{M}{8}$
- To be compute bound on my laptop with $I_m \approx 5$ requires $M \geq 40$
- This requires $40 \cdot 40 \cdot 8 \cdot 3 = 38400$ bytes. So 32 KBytes is pretty close
  Note: we need three matrices in the cache!

Other algorithms

- N-body problem: $O(n^2)$ operations on $O(n)$ data. Compute bound possible
- Discrete convolution:

$$(f * g)(i) = \sum_{j=-m}^{m} f(i-j)g(j)$$

- Effort is $O(nm)$, might be compute bound when m is large
- *Stencil computation* is a special case, there m is typically small ($m = 1$)

Programming Models

Overview

- SIMD Vectorization
  - Is mandatory for achieving high performance
- Thread programming
  - Great choice of programming models
  - OpenMP: simple when it works, C/C++/Fortran
  - C++ threads: multithreading in the standard but low-level/large effort
  - Intel thread building blocks: C++ lib, kernel-based and task-based, good scheduler
  - SYCL: Portable heterogeneous programming
- Message passing
  - MPI (message passing interface), now at version 4 is the model of choice
- MPI+X

How to use SIMD

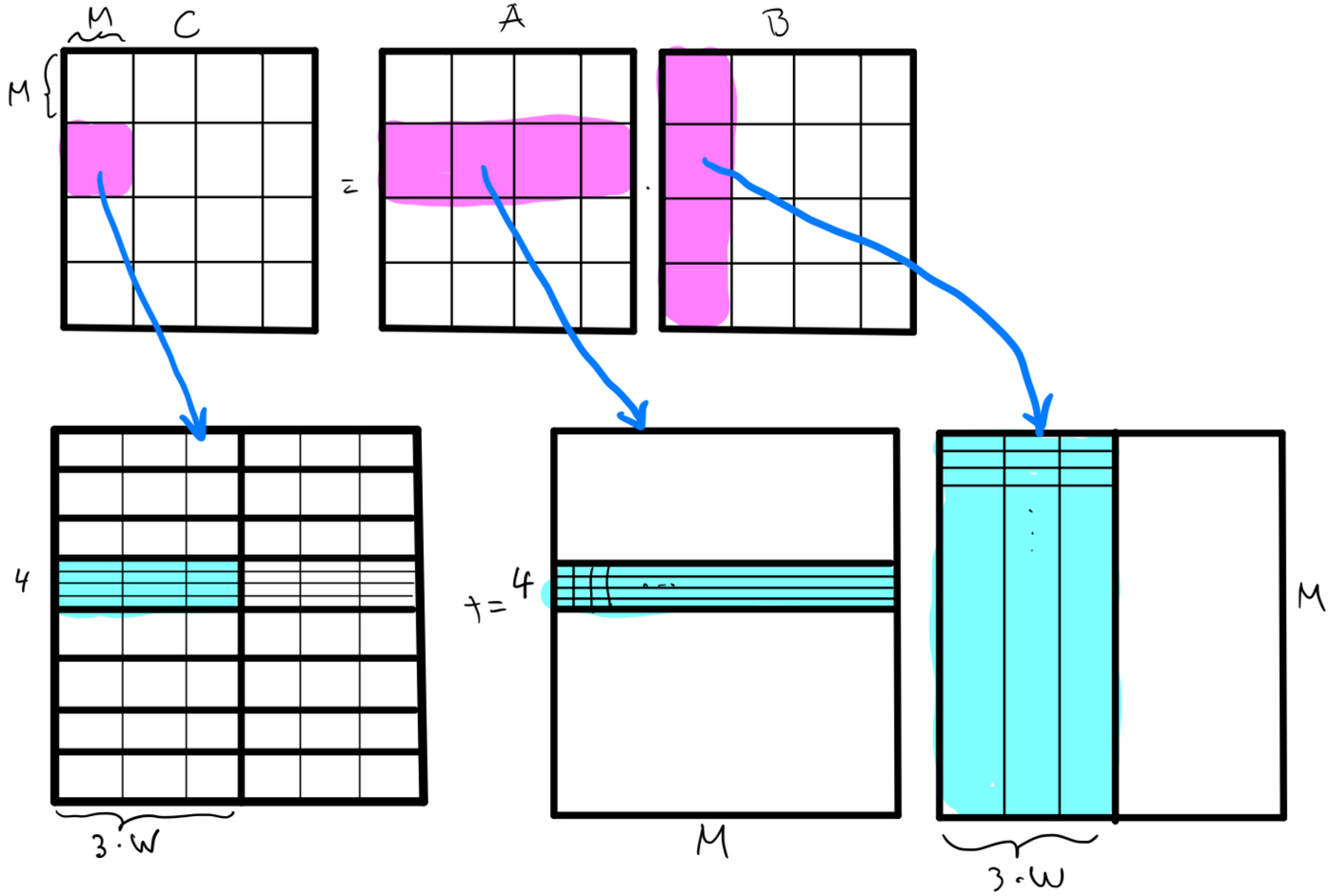
- Use compiler options
  - E.g. -mavx2 -mfma for gcc
- Write assembly language
- Intrinsics
  - Gcc (and other compilers) provide extensions with special data types, like __m256d for 4 doubles and functions on these types mapping directly to a single machine instructions. Its like assembler programming
  - Compiler does the register allocation and may do other optimizations (e.g. loop unrolling)
  - I found this article quite good http://const.me/articles/simd/simd.pdf
- Overloaded operators: A bit easier to use: packs these types into classes and overloads arithmetic operators. A bit more portable.
  - Example: Agner Fog's vector class library https://www.agner.org/optimize/

Scalability SIMD Vectorization

- Over the years SIMD width increased from 2 (SSE2) to 8 (AVX512) in DP
- Can we observe a corresponding performance increase?
- Can we write SIMD code in a generic way that supports different widths
- This might be important in the future with SIMD width up to 2048 bits in ARM CPUs (ARMv8 scalable vector extensions, used in Fujitsu A64FX CPU)
- Let us look at two examples: matmul and nbody
- C++ Implementation uses
  - Agner Fogs vector class library (overloaded operations map to intrinsics)
  - Templates to parametrize SIMD width at compile-time

Generic SIMD in matmul

- „4x3“ approach, SIMD width W, row-major storage
- Decompose $n \times n$ matrix into $M \times M$ blocks
- Each $M \times M$ matmul is decomposed into $4 \times (3W)$ blocks in C matrix, each such block requires multiplying a $4 \times M$ with a $M \times (3W)$ matrix using 12 FMA operations
- Required SIMD registers: 12 for C, 1 for A and 3 for B = 16!



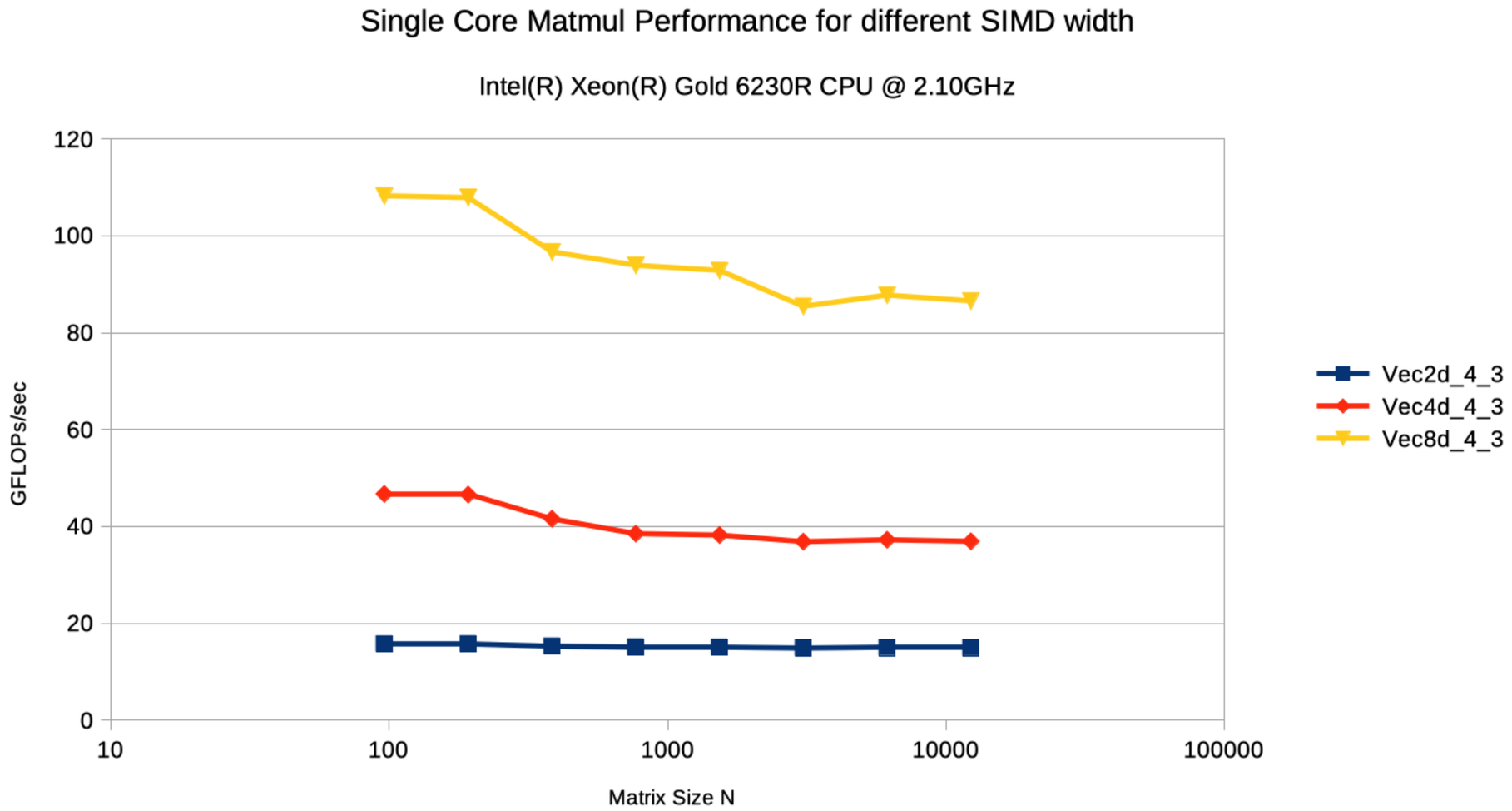
Generic SIMD Scaling Matmul Code

```
template<size_t simd_width>
struct SIMDSelector
{
};
template<>
struct SIMDSelector<2>
{
  static const size_t simd_width = 2;
  static const size_t simd_registers = 16;
  typedef Vec2d SIMDType;
};
template<>
struct SIMDSelector<4>
{
  static const size_t simd_width = 4;
  static const size_t simd_registers = 16;
  typedef Vec4d SIMDType;
};
template<>
struct SIMDSelector<8>
{
  static const size_t simd_width = 8;
  static const size_t simd_registers = 32;
  typedef Vec8d SIMDType;
};
```

```
// version with tile size and SIMD width as a parameter
// tiling and SIMD with vectorization of 4x3*W blocks
template<size_t M, size_t W>
void matmul4 (int n, double A[], double B[], double C[])
{
  using VecWd = typename SIMDSelector<W>::SIMDType;
  VecWd CC[4][3], BB[3], AA; // fits exactly 16 registers

  if (M%4!=0) {
     std::cout << "M must be a multiple of 4" << std::endl;
     exit(1);
  }
  if (M%(3*W)!=0) {
     std::cout << "M must be a multiple of 3*W" << std::endl;
     exit(1);
  }
  if (n%M!=0) {
     std::cout << "n must be a multiple of M" << std::endl;
     exit(1);
  }
```

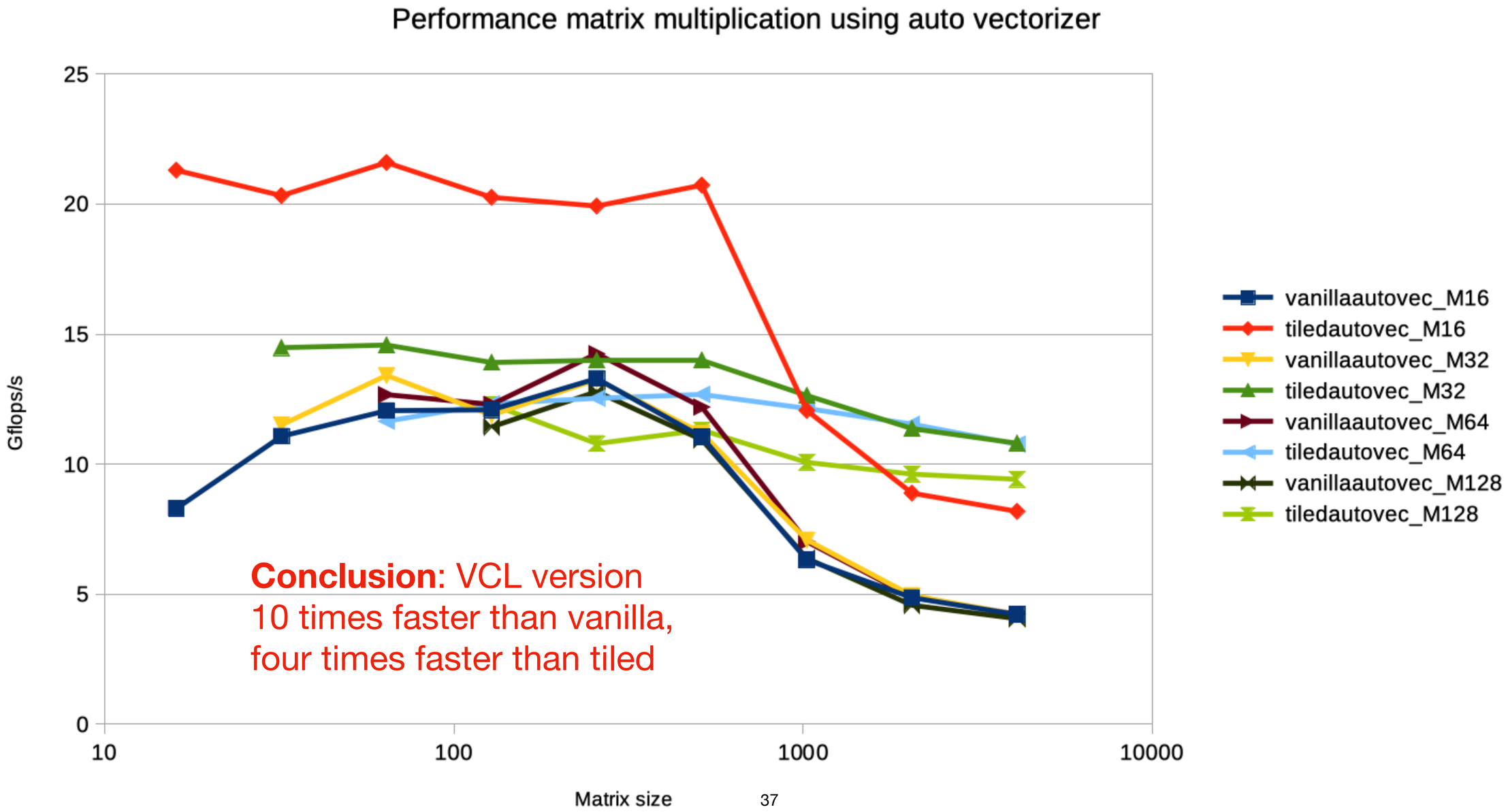
```
#pragma omp parallel for schedule (static) firstprivate(n,A,B,C) private(CC,BB,AA) collapse (2)
  for (int i=0; i<n; i+=M) // loop over tiles
    for (int j=0; j<n; j+=M)
      for (int k=0; k<n; k+=M)
        // C_ij += A_ik*B_kj where all blocks are MxM
        // now C_ij is again blocked into 4x(3*W) blocks
        for (int s=i; s<i+M; s+=4) // loop over 4x3*W blocks of C within the tiles
          for (int t=j; t<j+M; t+=3*W)
            {
              // C_st is a 4x3*W block in 12 SIMD registers which is loaded now
              for (int p=0; p<4; ++p)
                {
                  // load store amortized over M/8 matrix multiplications
                  CC[p][0].load(&C[INDEX(s+p,t,n)]);
                  CC[p][1].load(&C[INDEX(s+p,t+W,n)]);
                  CC[p][2].load(&C[INDEX(s+p,t+2*W,n)]);
                }
              // C_st += A_sM*B_Mt where now A_sM is 4xM and B_Mt is Mx3*W
              for (int u=k; u<k+M; u+=1) // columns of A / rows of B
                {
                  // 3 loads of B now amortized over ... 12 fmas
                  BB[0].load(&B[INDEX(u,t,n)]);
                  BB[1].load(&B[INDEX(u,t+W,n)]);
                  BB[2].load(&B[INDEX(u,t+2*W,n)]);

                  AA = VecWd(A[INDEX(s,u,n)]); // load-broadcast
                  CC[0][0] = mul_add(AA,BB[0],CC[0][0]);
                  CC[0][1] = mul_add(AA,BB[1],CC[0][1]);
                  CC[0][2] = mul_add(AA,BB[2],CC[0][2]);

                  AA = VecWd(A[INDEX(s+1,u,n)]); // load-broadcast
                  CC[1][0] = mul_add(AA,BB[0],CC[1][0]);
                  CC[1][1] = mul_add(AA,BB[1],CC[1][1]);
                  CC[1][2] = mul_add(AA,BB[2],CC[1][2]);

                  AA = VecWd(A[INDEX(s+2,u,n)]); // load-broadcast
                  CC[2][0] = mul_add(AA,BB[0],CC[2][0]);
                  CC[2][1] = mul_add(AA,BB[1],CC[2][1]);
                  CC[2][2] = mul_add(AA,BB[2],CC[2][2]);

                  AA = VecWd(A[INDEX(s+3,u,n)]); // load-broadcast
                  CC[3][0] = mul_add(AA,BB[0],CC[3][0]);
                  CC[3][1] = mul_add(AA,BB[1],CC[3][1]);
                  CC[3][2] = mul_add(AA,BB[2],CC[3][2]);
                }
              // write back C
              for (int p=0; p<4; ++p)
                {
                  // load store amortized over M/8 matrix multiplications
                  CC[p][0].store(&C[INDEX(s+p,t,n)]);
                  CC[p][1].store(&C[INDEX(s+p,t+W,n)]);
                  CC[p][2].store(&C[INDEX(s+p,t+2*W,n)]);
                }
            }
}
```

SIMD Scaling in Matmul



- Decent scaling with SIMD width and matrix size
- Probably turbo mode was on

For Comparison: Matmul results using auto vectorizer (AVX2)



Jacobi Method in 2d

- Example for memory bound algorithm

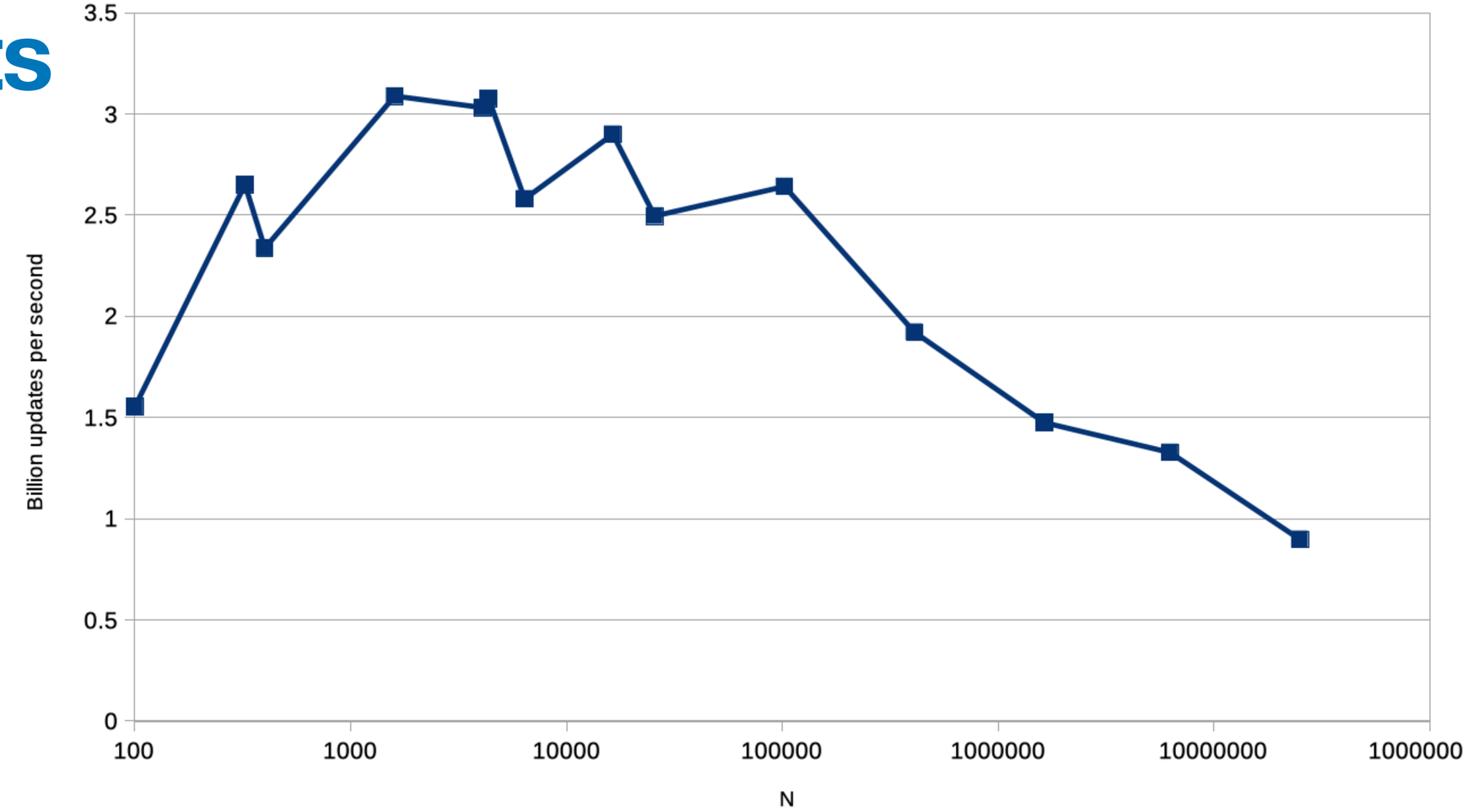
```
for k=1,…,iterations
  for i=1…n-1
    for j=1…n-1
```

$$u_{i,j}^{k+1} = \frac{1}{4}\left(u_{i-1,j}^{k} + u_{i,j-1}^{k} + u_{i+1,j}^{k} + u_{i,j+1}^{k}\right)$$

- Provide the initial guess in two vectors u0 and u1 and only update the interior
- Use double buffering to avoid a copy step
- No convergence check here, it would cost as much as one iteration!
- If you want one, amortize it over many iterations

```
void jacobi_vanilla (int n, int iterations, double* u0, double* u1)
{
  for (int i=0; i<iterations; i++)
    {
      for (int i1=1; i1<n-1; i1++)
        for (int i0=1; i0<n-1; i0++)
          u1[i1*n+i0] = 0.25*(u0[i1*n+i0-n]+u0[i1*n+i0-1]
                              +u0[i1*n+i0+1]+u0[i1*n+i0+n]);
      std::swap(u0,u1);
    }
}
```

Jacobi Results

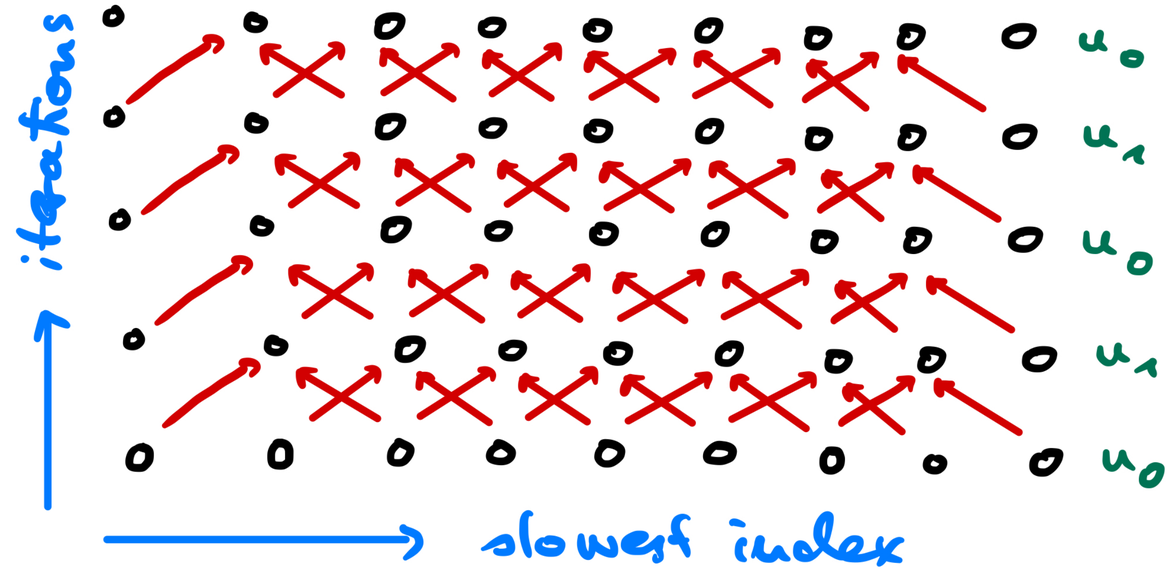


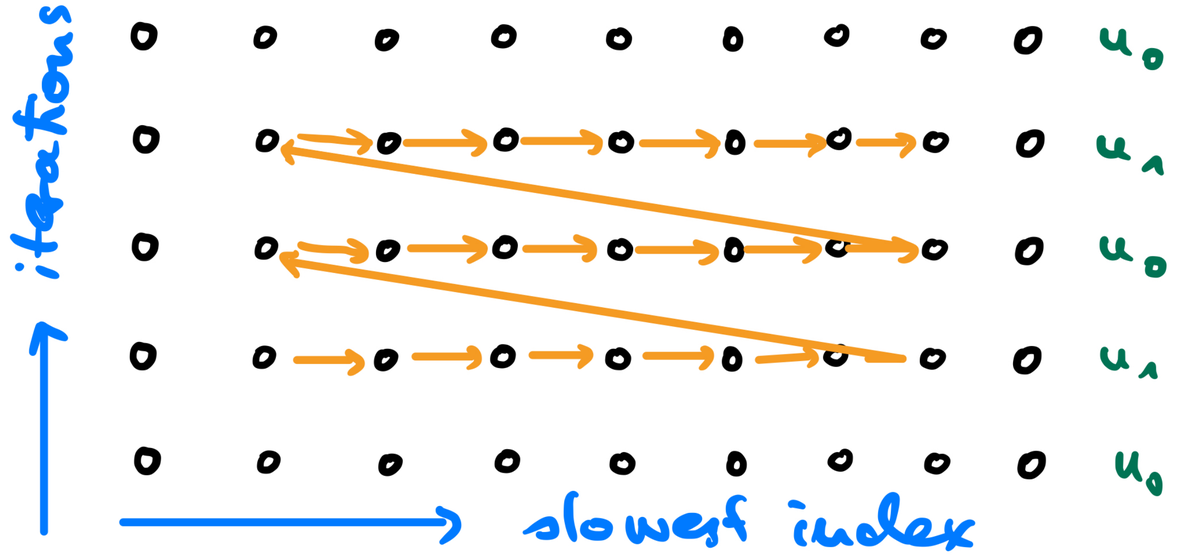
- X-axis shows total number of points in the lattice
- Y-axis gives (billion) site updates per second (1 update = 4 Flops)
- Compute intensity is 4/(6*8) = 1/12 (counting the cold write twice)

- N=1863225 fits 3 consecutive rows in L1 cache
- N=400000 fits both arrays completely into L3 cache
- N=16384 fits both arrays completely into L2 cache

Data Dependencies in 1d Jacobi

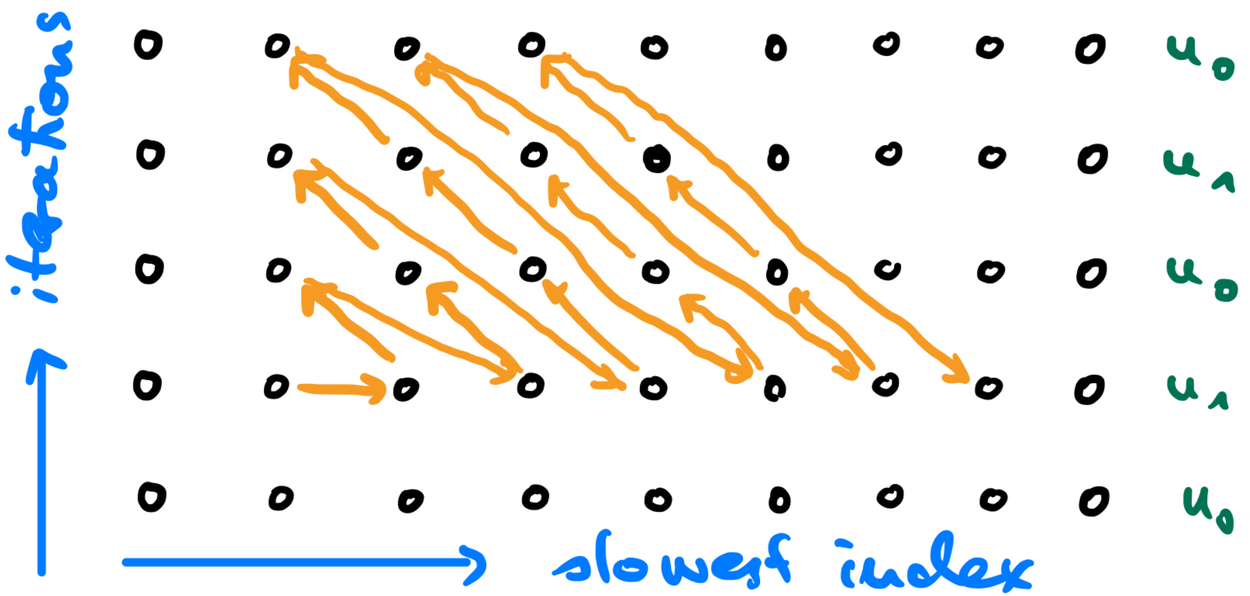
- Consider Jacobi in 1d
- Alternatively, each point represents a whole line (2d) or plane (3d)
- Standard scheme completes one iteration before the next starts

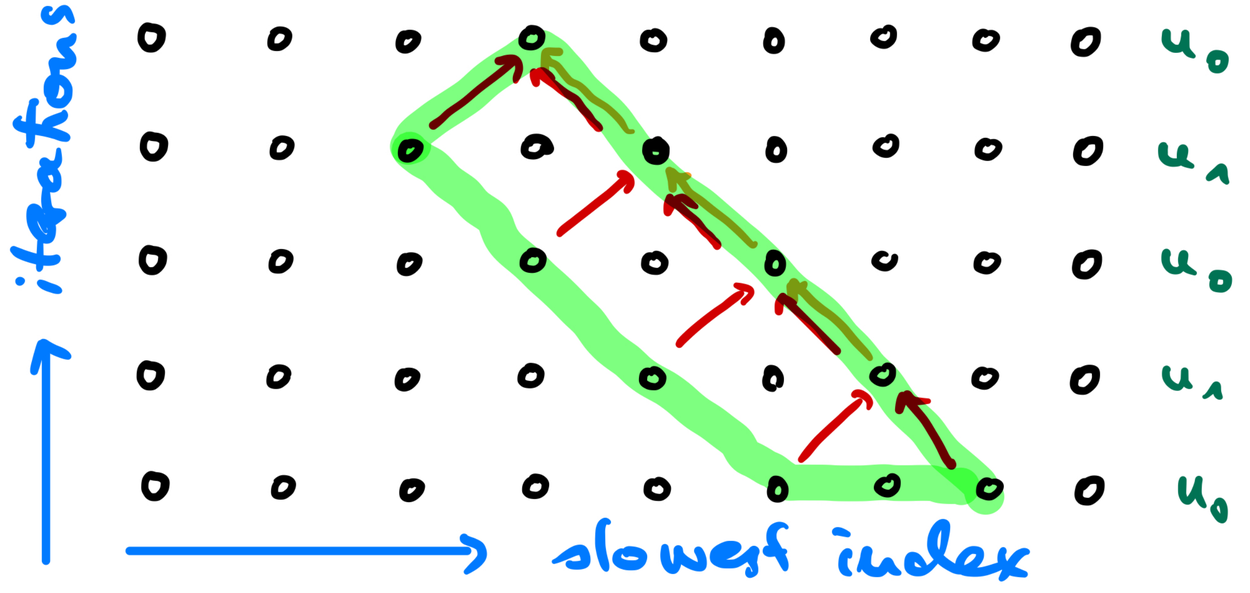




Wave Front Scheme

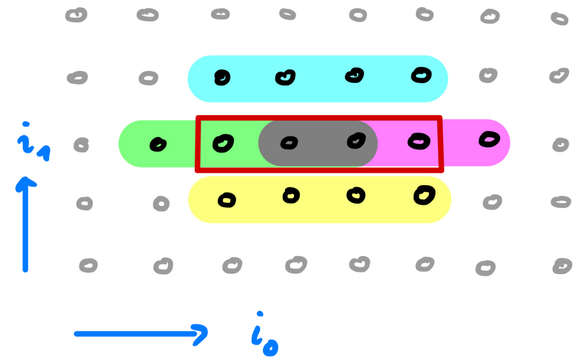
- Treat $K > 1$ iterations in an overlapping fashion
- Proceed in order shown to the right
- Uses both arrays K times for one spatial index (reading or writing)
- Need to fit data of about $3K$ spatial indices int cache
- Can you parallelize that scheme? Guess what!





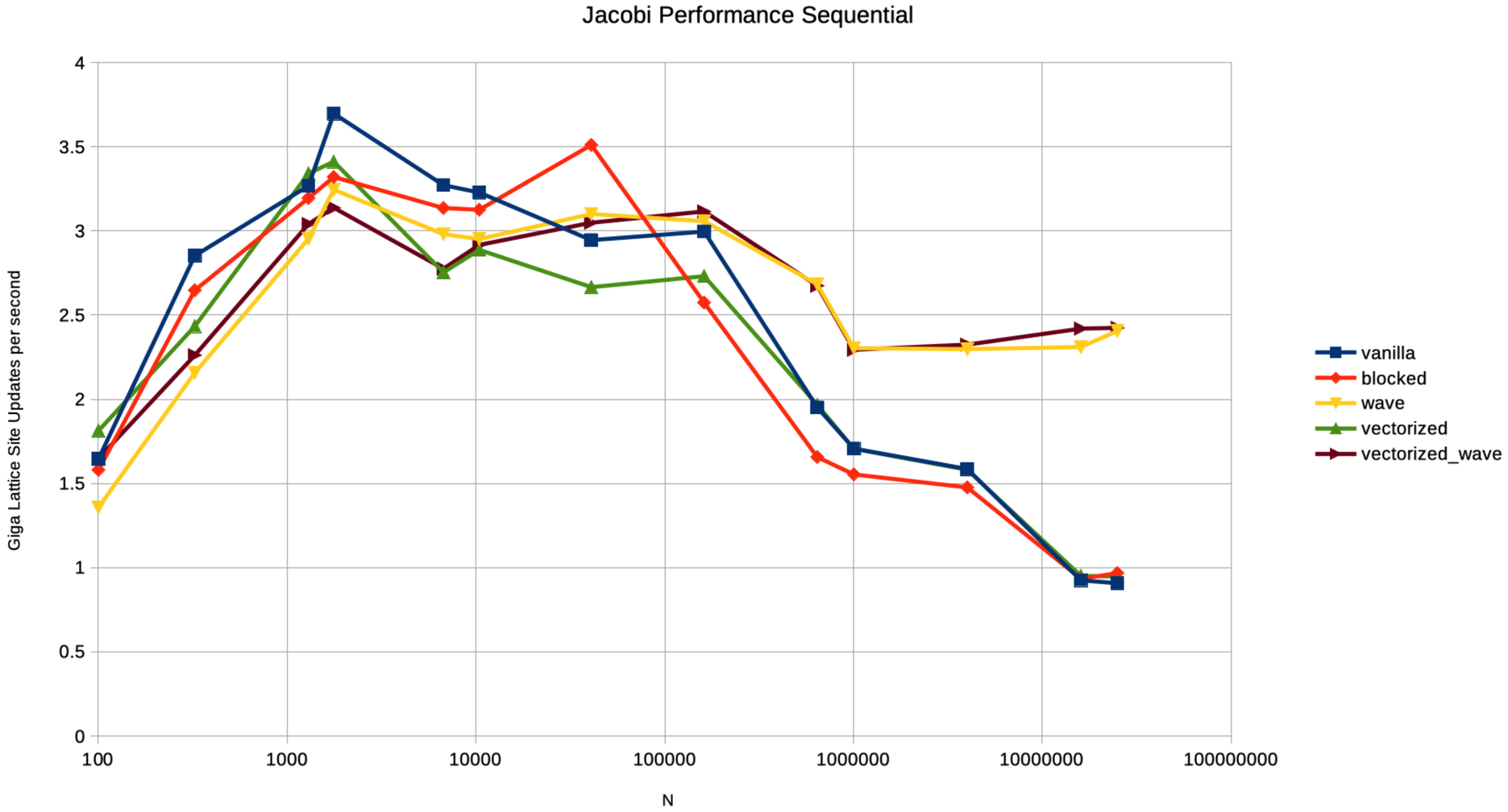
SIMD Vectorization

- Do not use horizontal adds, rather vectorize over 4 consecutive points in one row
- Needs four loads and one write
- Most loads will not be aligned
- FMA or not? Algorithm can be written with FMA
- This makes more flops and executes in the same time, so performance is higher
- But lattice updates per second stays the same
- Be careful with evaluating performance!




```
// do iterations
for (int i=0; i<iterations; i++)
  {
    for (int i1=1; i1<n-1; i1++)
      for (int i0=1; i0<n-1; i0++)
        unew[i1*n+i0] = 0.25*uold[i1*n+i0-n]
                      +0.25*uold[i1*n+i0-1]
                      +0.25*uold[i1*n+i0+1]
                      +0.25*uold[i1*n+i0+n]);
    std::swap(uold,unew);
  }
```

Improved Jacobi Performance



- Wavefront: 2.5*4=10 GFLOPS/sec in scalar/no fma or 2.5*8 = 20 GFLOPS/sec in vectorized FMA
- No performance advantage from vectorization

OpenMP Programming Model

- Is based on three components:
  - Compiler directives
    - Comments in Fortran
    - #pragma in C/C++ (a pragma gives additional information to the compiler beyond the language itself. The C standard specifies a few pragmas to be understood by every compiler, the rest is optional)
  - Runtime library
  - Environment variables
- Is quite easy to use as it avoids a lot of boiler-plate code for starting/joining threads, argument passing, return value passing
- Particularly simple for loop-based parallelism
- Fork-Join model: alternating sequential and parallel phases executed by a team of threads
- Newer versions support also task-based parallelism and SIMD vectorization

Hello World Example

```
#include <iostream>

#ifdef _OPENMP
#include<omp.h> // headers for runtime if available
#endif

int main (int argc, char** argv)
{
  // start sequential as usual
#pragma omp parallel // execute the following block in parallel
  { // number of parallel threads conrolled in various ways
    int id = omp_get_thread_num(); // call library function
#pragma omp critical // execute following block exclusive
    std::cout << "I am " << id << std::endl;
  } // join parallel threads at end of parallel regions
  return 0;
}
```

- Compile with option -fopenmp in g++

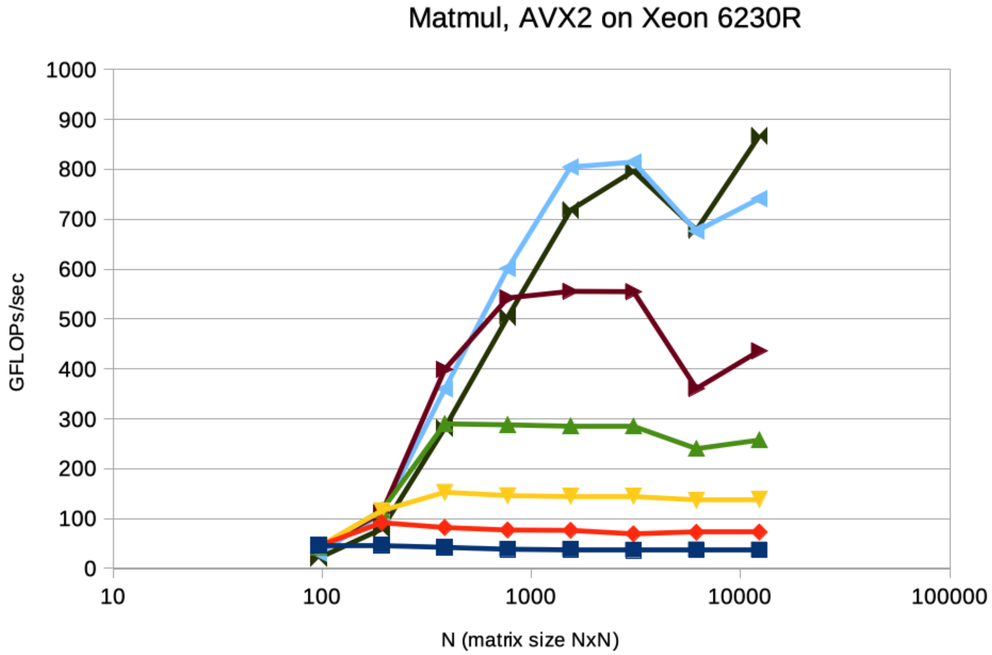
Example: Scalar Product

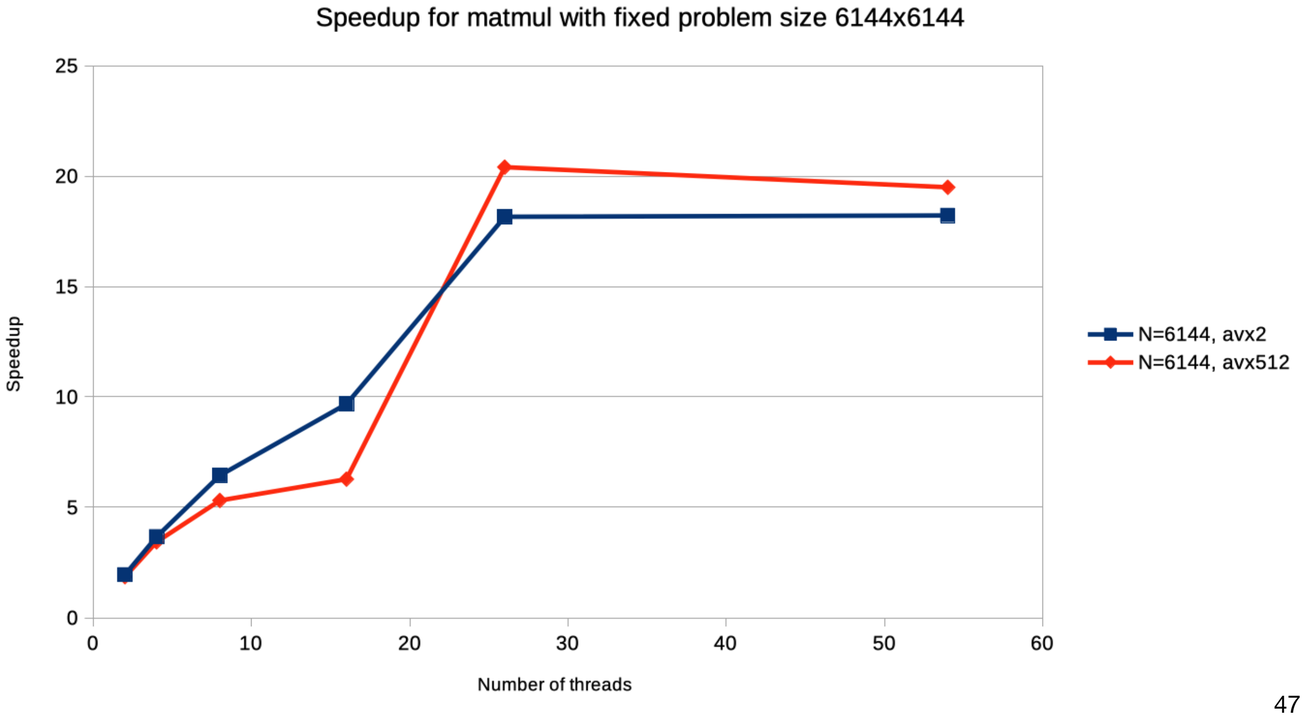
- result is private in the parallel section and reduced at the end

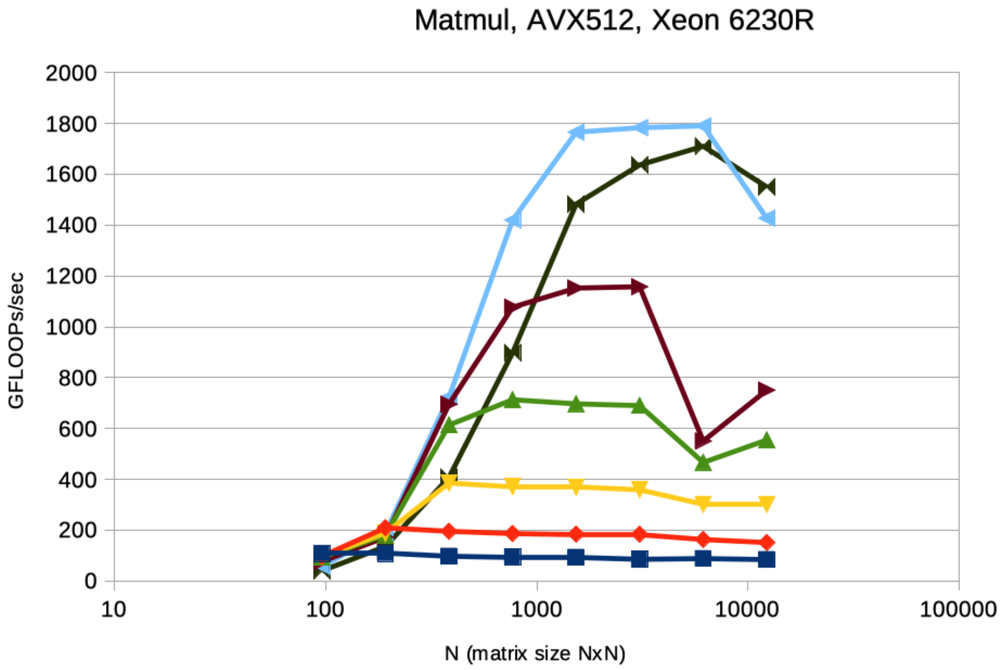
```
  // Example: parallel scalar product
  const int n=100;
  double x[n];
  double y[n];
  double result=0.0;
  for (int i=0; i<n; i++) x[i]=1.0/(1.0+i);
  for (int i=0; i<n; i++) y[i]=(1.0+i);

  // scalar product parallel for loop
#pragma omp parallel for \
  num_threads (4) \
  schedule (static) \
  shared (x,y,n) \
  reduction (+: result)
  for (int i=0; i<n; i++) result += x[i]*y[i];
  std::cout << "scalar product is "
            << result << std::endl;
```

OpenMP Matmul

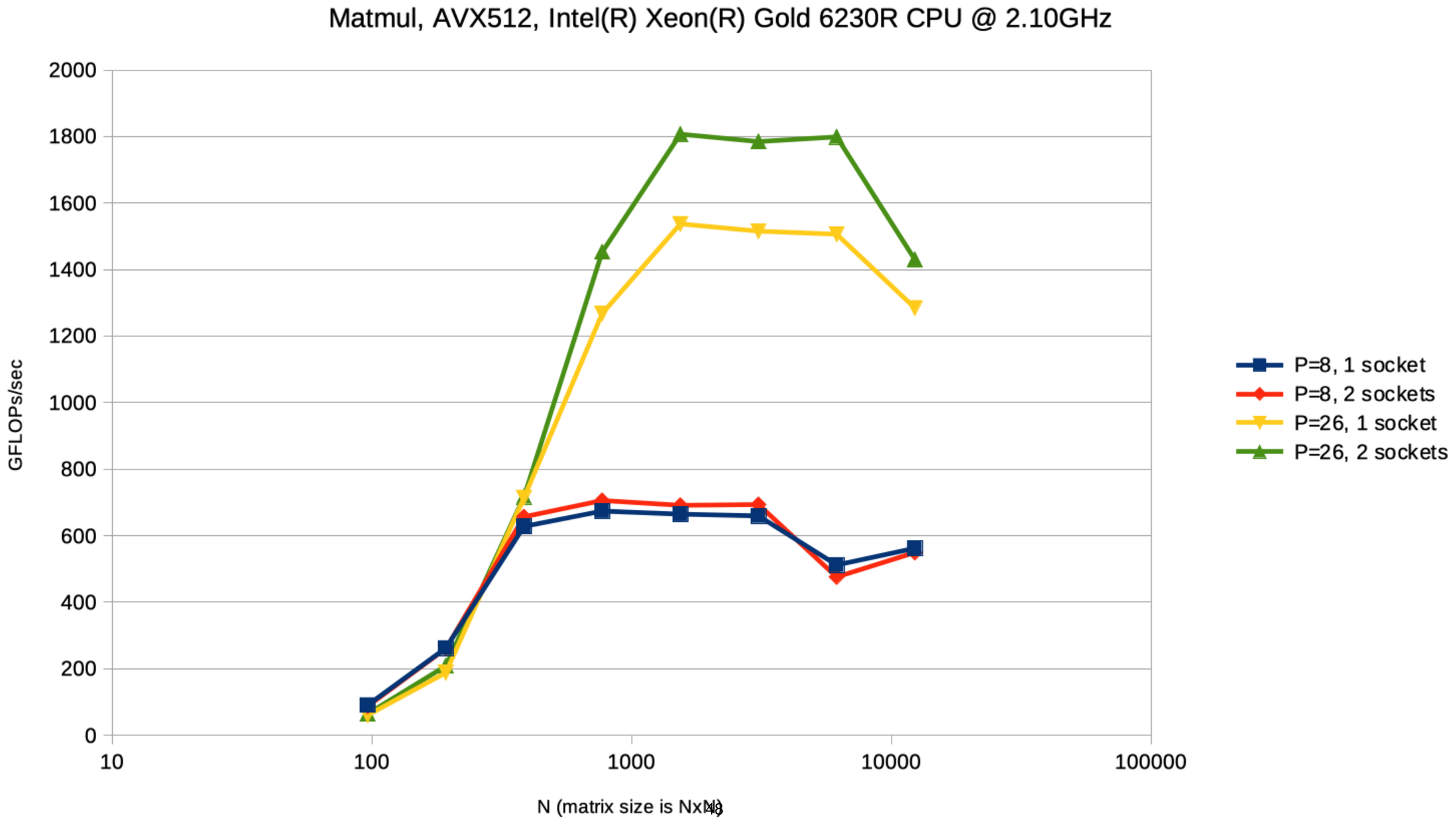
- 2 x Xeon Gold 6230R CPU
- Has 2x26 cores with AVX512







Effect of Pinning for Matmul



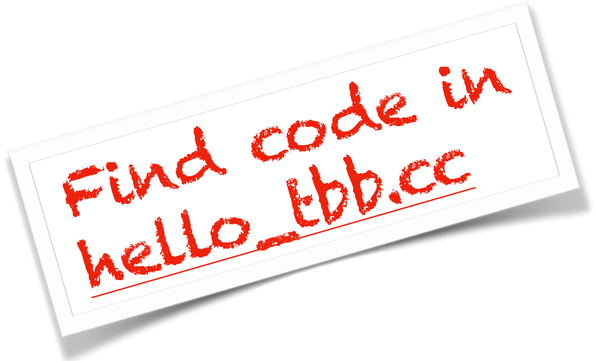
TBB Intro

- Open-source C++ library supporting thread-level shared memory parallelism on a single node since 2006
- Central concepts: *kernels* and *tasks*
- It is now part of Intel *oneAPI*, a new attempt to provide an open platform for writing code on heterogeneous systems incorporating accelerators (GPUs, CPUs, FPGAs), oneAPI contains
  - Compilers: *icc* (classic Intel compiler) and *dpcpp* (data parallel C++), both compile *OpenMP* code
  - *TBB* which you can compile with any compiler
  - *SYCL*, a data parallel programming language and run-time library (dpcpp compiler only)
  - Intel *MPI* implementation
- Note for MacOS users: dpcpp is not supported on MacOS :-(
- There is an excellent free book on TBB: https://www.springer.com/de/book/9781484243978
- Documentation for TBB can be found here: https://software.intel.com/content/www/us/en/develop/documentation/onetbb-documentation/top.html

Complete Vector Addition Example

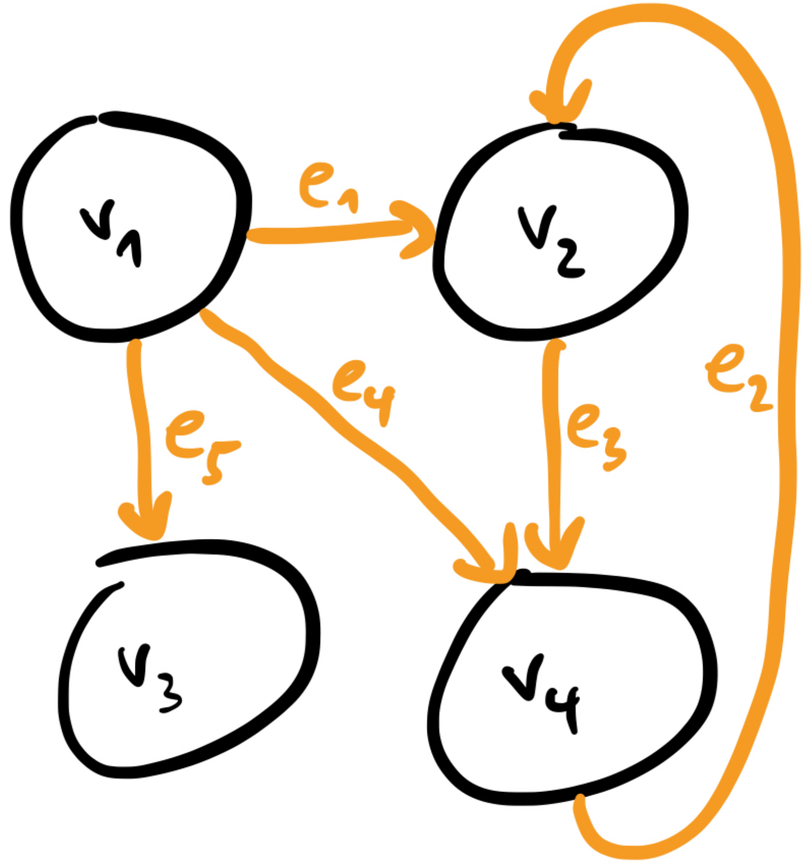
```
#include <iostream>
#include <vector>
#include <oneapi/tbb.h>

class VectorSumKernel
{
  std::vector<double> &x,&y,&z;
public:
  VectorSumWorker (std::vector<double>& _x, std::vector<double>& _y,
                   std::vector<double>& _z)
    : x(_x), y(_y), z(_z)
  {}
  void operator() (const oneapi::tbb::blocked_range<size_t>& r) const
  {
    for (size_t i=r.begin(); i<r.end(); ++i) z[i]=x[i]+y[i];
  }
};

int main (int argc, char** argv)
{
  std::vector<double> x(1000,1.0), y(1000,2.0), z(1000);

  // first version: pass kernel object
  oneapi::tbb::parallel_for(oneapi::tbb::blocked_range<size_t>(0,x.size()),
                            VectorSumKernel(x,y,z));

  // second version: pass lambda
  oneapi::tbb::parallel_for(oneapi::tbb::blocked_range<size_t>(0,x.size()),
                            [&](const oneapi::tbb::blocked_range<size_t>& r)
                            {
                              for (size_t i=r.begin(); i<r.end(); ++i)
                                z[i]=x[i]+y[i];
                            });

  return 0;
}
```

- parallel_for is used to invoke a kernel function for a given range
- In addition, a chunk size parameter could be given
- Observe the similarity to parallelism in the C++ standard library



Data Flow Graphs

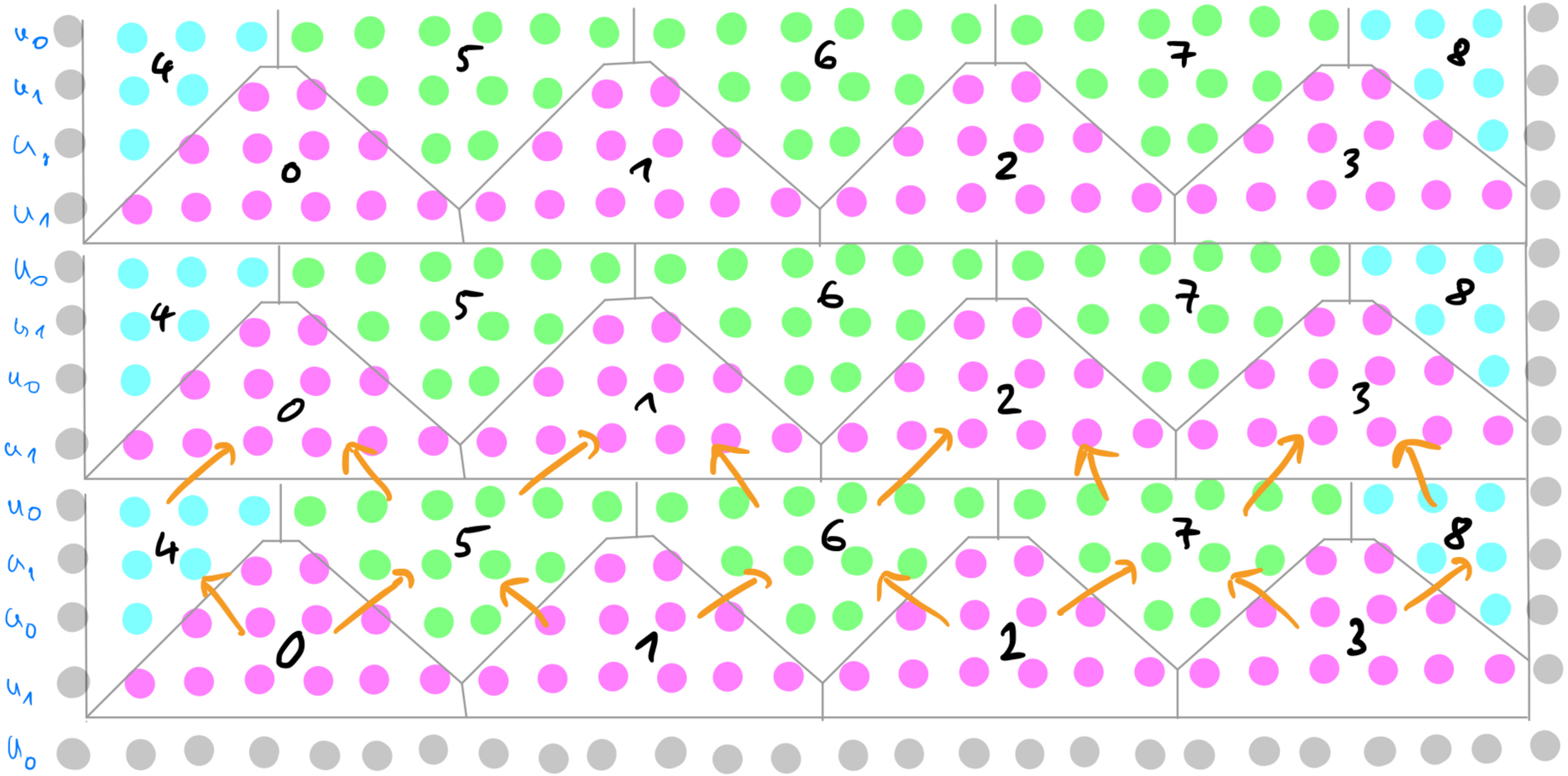
- We now turn to a different parallelization approach: data flow graphs
- Given a *directed* graph $G = (V, E)$ consisting of nodes and edges
- **Each node stands for a computation** working on input data given by the input edges and producing a result corresponding to the output edges
- **Each edge stands for a message** transferred from the source node to the destination node
- Consider e.g. node v_2, it realizes a computation $f_2 : T_1 \times T_2 \to T_3$, where T_i is the type transferred on edge e_i
- The computation is carried out as soon as an input message is available on *every* input edge
- Data flow graphs may be cyclic!



Data Dependence Graphs

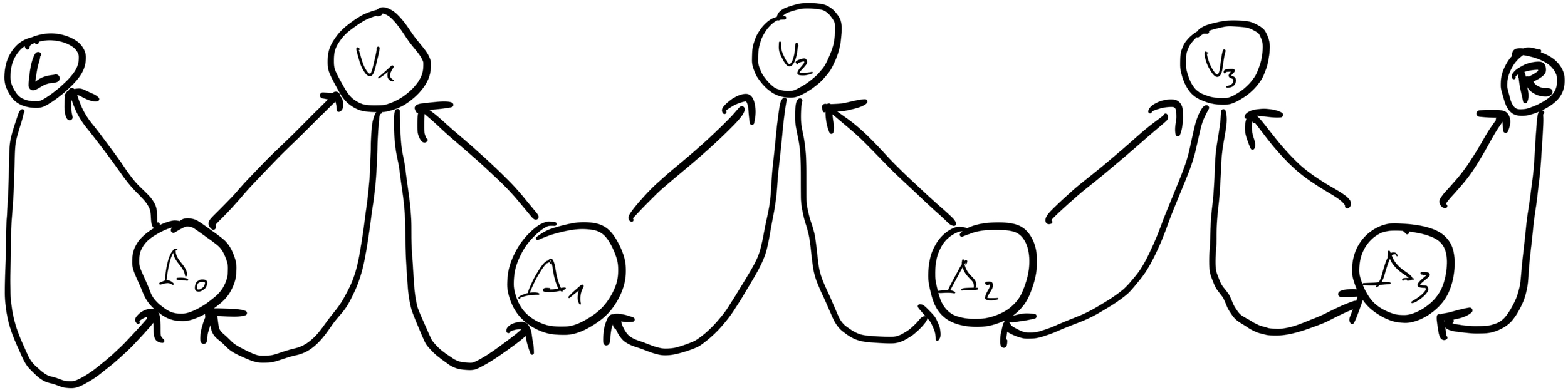
- *Pure data flow graph*: computation only depends on the inputs provided by the input messages
- *Data dependence graph*:
  - Computation is performed on some shared state
  - The input messages provide synchronization and determine *when* the computation can be done
- TBB provides the continue_node for that, using a dummy message type continue_msg (but you could do it on your own as well)
- Data dependence graphs realized with continue_node *may not be cyclic*!
- Below we use mixed form of a data dependence graph:
  - Computations are performed on a global shared state
  - Messages convey iteration numbers; they determine when the next iteration can be started and when the overall computation is finished
  - In this way, also cyclic data dependence graphs can be covered!

Application: Parallel Jacobi



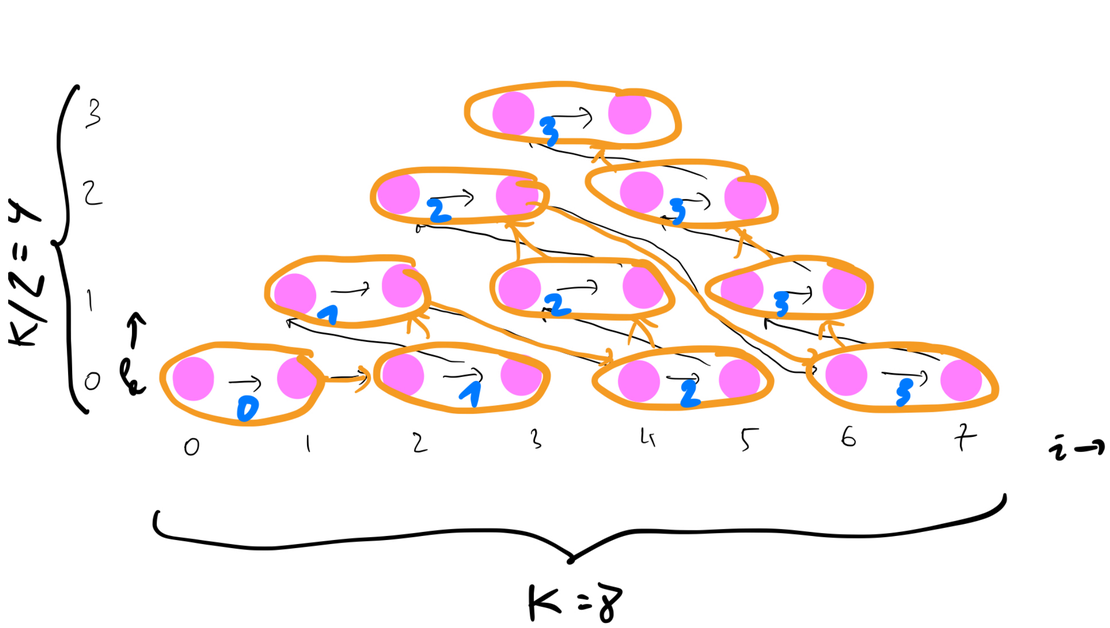
- Good performance needs data locality across iterations
- Figure: horizontal: i_1 index, vertical: iterations
- Let K be an even integer defining groups with horizontal extend K and vertical extend $K/2$
- Groups 0,1,2,3 can be computed in parallel; orange edges indicate data depend

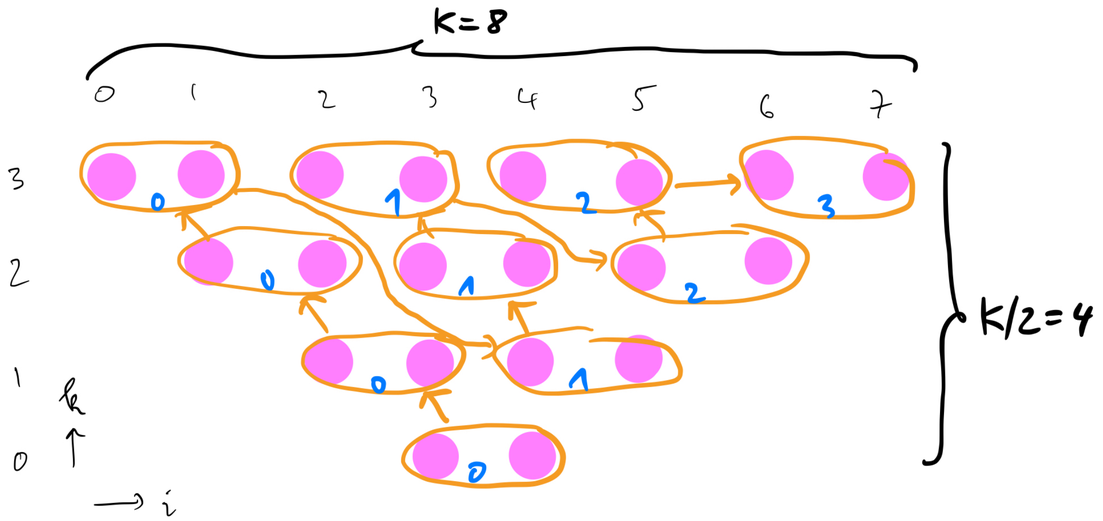
Conceptual Data Dependence Graph



- Λ nodes compute the Lambda-shaped triangular regions
- V nodes compute the V-shaped triangular regions
- L, R compute the left and right triangular regions of half size

Processing the Chunks





- Processing order in each chunk is chosen to achieve good locality

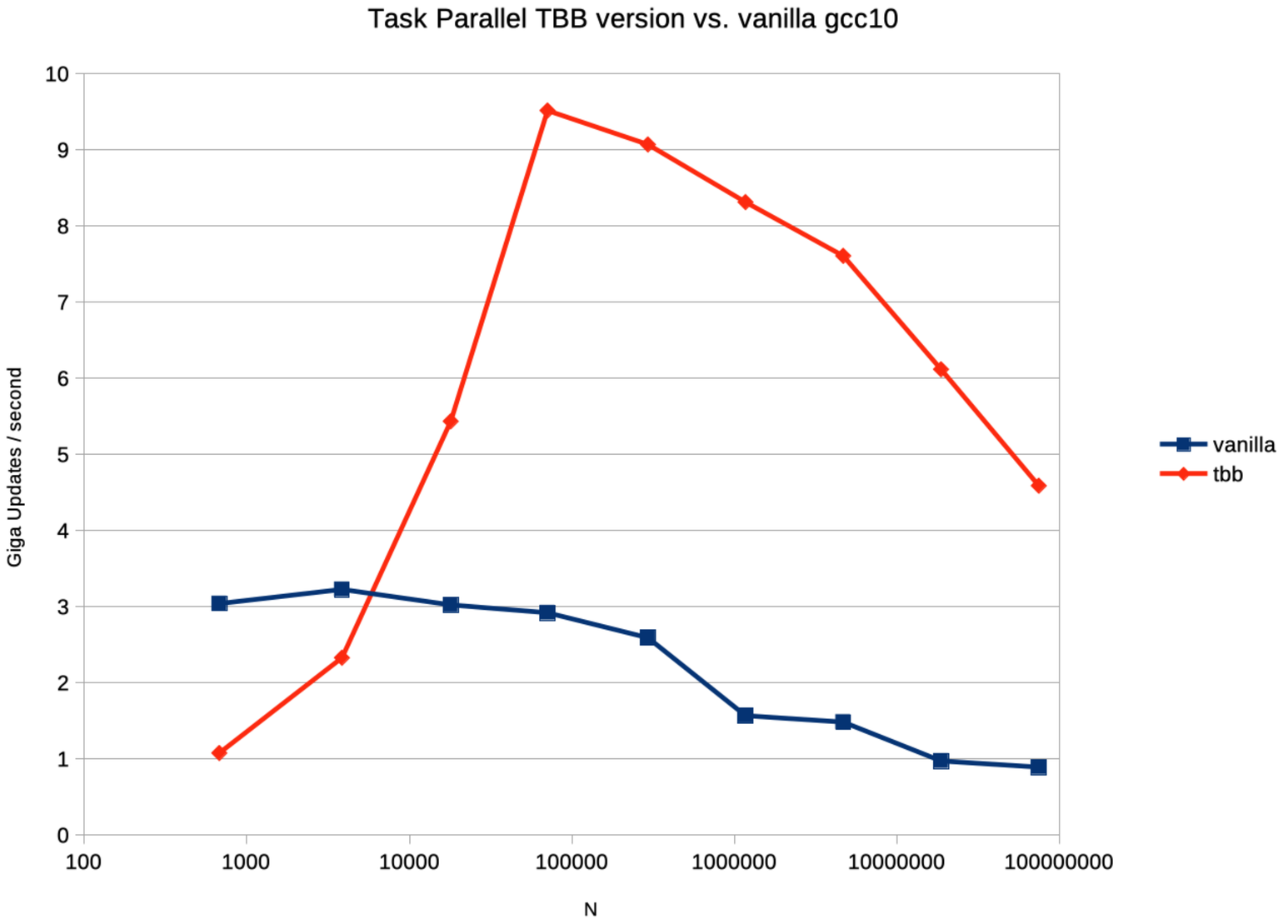
Λ Node

- Λ nodes are normal function nodes
- The function to be performed is the operator()
- The constructor gets grid size n, the two arrays to work on and the chunk number i to identify itself
- Input type is std::tuple<int,int> which is the output of the preceding join_node. Each int is the number of the iteration we are in (starting with 1)
- Output type is one int (the same iteration)
- The iteration number is increased in the L, R, V nodes
- Actually each iteration stands for a group of $K/2 + 1$ iterations (we assume, the total number of iterations is a multiple thereof)

```
using Lambda_node = oneapi::tbb::flow::function_node<
  std::tuple<int,int>,int>;

template<int K>
class Lambda
{
  int n;
  double* u[2];
  int i;
  int i1start;
public:
  Lambda (int _n, double* u0, double* u1, int _i)
    : n(_n), i(_i)
  {
    u[0] = u0; u[1] = u1;
    i1start = 1+i*K;
  }
  int operator() (const std::tuple<int,int>& in)
  {
    for (int r=0; r<K/2; ++r)
      for (int k=0; k<=r; ++k)
        {
          int src = k%2;
          int dst = 1-src;
          int i1 = i1start+(r-k)*2+k;
          for (int i0=1; i0<n-1; i0++)
            u[dst][i1*n+i0] = 0.25*(u[src][i1*n+i0-n]
                                    +u[src][i1*n+i0-1]
                                    +u[src][i1*n+i0+1]
                                    +u[src][i1*n+i0+n]);
          i1++;
          for (int i0=1; i0<n-1; i0++)
            u[dst][i1*n+i0] = 0.25*(u[src][i1*n+i0-n]
                                    +u[src][i1*n+i0-1]
                                    +u[src][i1*n+i0+1]
                                    +u[src][i1*n+i0+n]);
        }
    if (K/2%2==0) std::swap(u[0],u[1]);
    return std::get<0>(in);
  }
};
```

Find code in jacobi_tbb.cc

Jacobi Performance



Wrap Up

- Todays CPUs are highly parallel, complex machines
- Peak performance can only be achieved by
  - Using SIMD instructions
  - Using multiple cores
- Use roofline analysis to think about your algorithm
- GPUs:
  - SIMD on steroids (wider, more and only)
  - High bandwidth memory