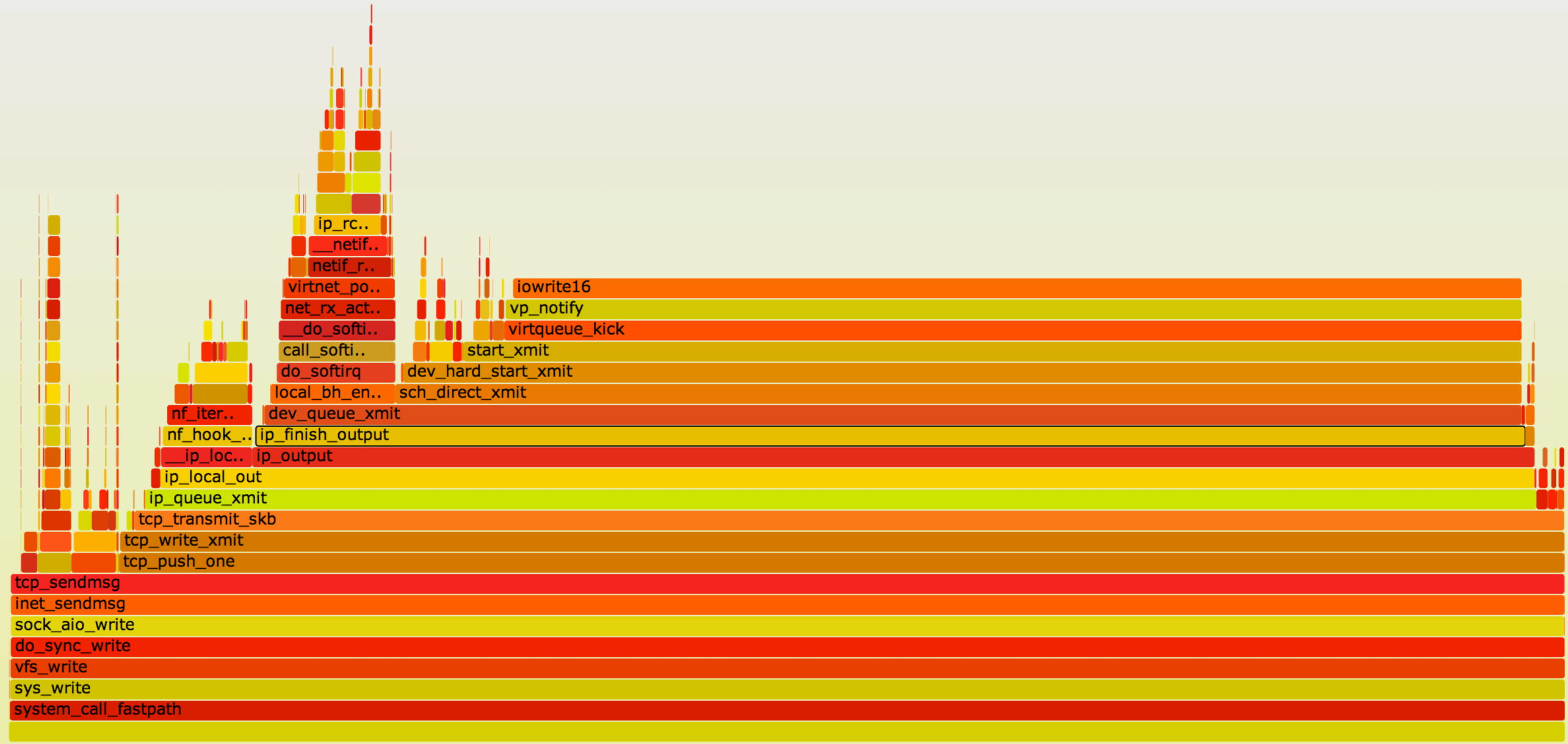


Debugging and profiling

Introduction to executable binaries

Antonin Portelli

Supported by UK EPSRC grants EP/V001329/1, EP/V001191/1, EP/V001477/1, EP/V001248/1, EP/V001140/1

- Crash debugging
- Memory debugging
- Profiling and optimisation
- Wrap up

Crash debugging

Abnormal program terminations

- An abnormal program termination (crash) is when a program was interrupted before reaching an expected end
- There are many ways this can happen
- Maybe the program did something illegal and the OS killed it (asked too much memory, illegal access, …)
- Maybe the program interrupted itself because it detected an anomalous behaviour with no path to correct
- Maybe the program just froze, and the user manually interrupted it

Signals

- When a program crashes, it emit a **signal** and returns a non-zero value, possibly with the same reference

- The **signal number** is generally reported by whatever is handling the program (OS, MPI, debugger, …)

- Also in the shell variable `$?` after the crash

- Look at it! It provides useful information on what happened

Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from abort(3)
SIGALRM	P1990	Term	Timer signal from alarm(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for **SIGCHLD**
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for **SIGPWR**
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for **SIGABRT**
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL	P2001	Term	Pollable event (Sys V); synonym for **SIGIO**
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also seccomp(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	-	Core	Synonymous with **SIGSYS**
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see setrlimit(2)
SIGWINCH	-	Ign	Window resize signal (4.3BSD, Sun)

Standard UNIX signals

Catching signals

- Signal **can be caught** during the execution using the standard C function **`signal`**. A **custom function** (signal handler) is executed is the signal is caught

- Signal can be sent using the **`kill`** command. Except for some special signal like SIGKILL, it actually does not kill the program!

- Handling signals allow to **potentially recover** from issues

- It also allow to **display more information** about the encountered issue (e.g. backtrace)

Catching signals

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void signal_handler(int signum)
{
    printf("Exiting with signal %d\n", signum);
    exit(signum);
}

int main(void)
{
    int s;

    // set handler for all signals
    // warning: _NSIG is potentially Linux-only
    for (s = 0; s < _NSIG; ++s)
    {
        signal(s, signal_handler);
    }
    // infinite loop waiting to be interrupted...
    while (1)
    {
        printf("Doing nothing...\n");
        sleep(1);
    }

    return EXIT_SUCCESS;
}
```

C++ exceptions

- Exception are a high-level construct allowing to report and handle abnormal events in a C++ program.

- An exception is a full user-defined object that can be **thrown** and then **caught** anywhere in the execution.

- Exceptions are a form of **non-local** communication

- If an exception is not caught, **the program aborts** by calling `std::terminate()`

- Exceptions are **very different from signals**, even if caught the original execution stack **cannot be resumed**

Using the debugger

- GDB & LLDB have **automatic breakpoints** in function handling crashes (signals and exceptions)

- Running a problematic program within a debugger is a **quick and easy way to get a backtrace of the issue**!

- Once you have a first backtrace, you can create more breakpoints around the problem location and investigate more when the execution is paused

- Debuggers are not called debuggers without reasons. They give you a **systematic way to eliminate even extremely non-trivial bugs**

“Coding without a debugger is like walking with your eyes closed or driving at night with no headlights.

Sure, it may be possible, but you are purposefully limiting your information in order to not become dependant on something.”

– **some internet person**

Post-mortem analysis, coredumps

- Linux can save a snapshot of the machine when a crash occurs. It's called a **core dump**.
  Activate it with `ulimit -c unlimited`

- Core dumps can be loaded in LLDB with `lldb -c core`.
  You can explore everything (memory, registers, backtrace) at the moment of the crash. This is extremely useful!

- Limitation: **you can't step** in the program (makes sense)

- Core dumps are ELF files, you can use `readelf` and `objdump` on them

- As always, read the man page https://bit.ly/37Cgnwu

Segmentation fault resolution

Segmentation fault resolution

Dont’ panic.

90% of the segmentation faults I have seen came from trivial mistakes.

You just need to be methodical to locate the issue.

The 10% remaining, well…

Segmentation fault resolution

- Segmentation fault mainly means that the program tried to access memory outside its segments

- It can happen for a **very wide spectrum of scenarios**… **Don’t over-interpret it** without additional evidence

- In any case it should always start by

  1. Establish if the error is **reproducible** and how

  2. Get a **backtrace** of the error

- Then debug around the found location directly in the program or using a core dump

Example: segmentation fault core dump

- Simple one, do you see it?

```c
int main(int argc, char **argv)
{
    unsigned int l;
    char         *s = argv[1];

    for (l = strlen(s); l >= 0; l--)
    {
        putchar(s[l]);
    }
    putchar('\n');

    return EXIT_SUCCESS;
}
```

- Debug session from core dump https://bit.ly/2ZWa7ev

Memory debugging

Why memory bugs are your worst enemy

- They can be **hard to reproduce** because of side effects
- They can be **“quantum”**: disappear when being debugged or if optimisation is being reduced
- They can **silently corrupt data**, terrible for scientific work
- They can be **exploited** to hijack a program execution, less relevant for scientific applications, critical for security
- **Always check boundaries at the interface level**
  Always check return values of memory functions

Buffer overflows

- Buffer overflow happens when a program access an array **outside of its allocated range**

- This is **not strictly speaking illegal**, the accessed memory might very well be part of the program memory

- High-level languages (e.g. Python) are **protected against buffer overflows**, i.e. array indices are always checked

- In C/C++ it is perfectly legal to go beyond an array boundary. C++ offers bound-checked access functions (cf. e.g. `std::vector::at` and `std::map::at`)

More about buffer overflows

- Buffer overflows are **very serious**. *At best* they will just crash the program, but they can silently corrupt memory and are security vulnerabilities

- If somebody tells you buffer overflows exist in C/C++ because they are old languages, don't believe them!

- C/C++ assumes (almost) no implicit behaviour, **it is the programmer's job to know how to handle the memory**

- **This is a feature, not an issue**. Systematically checking array boundaries can be more expensive than accessing the array itself! (think about instructions needed in ASM…)

Buffer overflow example

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```
$ ./boverflow aaaaaaaaa
Starting Fibonacci sequence with label aaaaaaaaa
1 1 2 3 5 8 13 21 34 55 89 144
$ ./boverflow aaaaaaaaaaaaaaaaaaaa
Starting Fibonacci sequence with label aaaaaaaaaaaaaaaaaaaa
1 1 2 3 5 8 13 21 34 55 89 144
$ ./boverflow aaaaaaaaaaaaaaaaaaaaaaaaaaa
Starting Fibonacci sequence with label aaaaaaaaaaaaaaaaaaaaaaaaaaa
1 6381921 6381922 12763843 19145765 31909608 51055373 82964981 134020354 216985335 351005689 567991024
$ ./boverflow aaaaaaaaaaaaaaaaaaaaaaaaaa
Starting Fibonacci sequence with label aaaaaaaaaaaaaaaaaaaaaaaaaa
1 24929 24930 49859 74789 124648 199437 324085 523522 847607 1371129 2218736
$ ./boverflow aaaaaaaaaaaaaaaaaaaaaaaaa
Starting Fibonacci sequence with label aaaaaaaaaaaaaaaaaaaaaaaaa
1 97 98 195 293 488 781 1269 2050 3319 5369 8688
$ ./boverflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Starting Fibonacci sequence with label aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
1633771873 1633771873 -1027423550 606348323 -421075227 185273096 -235802131 -50529035 -286331166 -336860201 -623191367 -960051568
[1]    33243 segmentation fault (core dumped)  ./boverflow
```

```c
    }
    printf("\n");

    return EXIT_SUCCESS;
}
```

Buffer overflow example

- The buffer for the label can only contain 16 characters

- It is on the stack **above** the initial conditions `x0` & `x1`

```
main:
  push rbp
  mov rbp, rsp
  sub rsp, 80
  lea rax, [rbp - 48]            ; buffer, 48B from bottom of stack
  mov dword ptr [rbp - 4], 0
  mov dword ptr [rbp - 8], edi
  mov qword ptr [rbp - 16], rsi
  mov dword ptr [rbp - 20], 1    ; x0, 28B under buffer
  mov dword ptr [rbp - 24], 1    ; x1, 24B under buffer
  mov rcx, qword ptr [rbp - 16]
  mov rsi, qword ptr [rcx + 8]
  mov rdi, rax                   ; rax = buffer
  ; ...
```

- Feeding more than 16 characters **overwrite the initial conditions** and modify the sequence

- It **happens silently** and the behaviour changes if you declare variables in another order!

Memory leaks

- Memory leaks occur when heap-allocated memory is **not properly freed** when its content is not used anymore
- In C/C++ the heap management is **left to the developer**
- Again, this is a feature, **heap allocations are expensive**
- Maybe less severe than buffer overflows, but can cause program to **unexpectedly run out of memory** and crash
- **Encapsulate heap allocations** in higher-level logics to reduce the risk of introducing leaks
- In C++, use **smart pointers** and **containers** as much as you can (they have automated heap management)

Memory debuggers

- Memory debuggers (or profilers) are tools that allow the user to **analyse the memory handling of their program**

- They sometime achieve that by **overriding allocators** by safer one, it can have a severe effect on performances

- Some tools allow to browse the **table of allocations**

- Some tools allow to **detect anomalous events**, like probable leaks or out-of-range accesses

- It is **healthy to pass your software through such tool** on a regular basis

LLVM address sanitiser

- The LLVM **address sanitiser** can instrument a code to detect leaks, buffer overflows, and other memory errors

- It is used by passing the option `-fsanitize=address` to clang at compile time

- LLVM proposes a number of other useful sanitisers, they all **significantly affect performances** and should not be used to produce production binaries

- Errors are reported during the program executions

- Documentation https://bit.ly/3aHKXqa

LLVM address sanitiser demonstration

- Simple C memory leak https://bit.ly/301nk5Q

```
$ clang -fsanitize=address -fno-omit-frame-pointer -g leak.c -o leak
$ ./leak

=================================================================
==30750==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 80 byte(s) in 1 object(s) allocated from:
    #0 0x493a1d in malloc (/mnt/hgfs/binary-hack/lectures/3-debug-profile/leak+0x493a1d)
    #1 0x4c312b in make_vector /mnt/hgfs/binary-hack/lectures/3-debug-profile/leak.c:7:22
    #2 0x4c31bc in swap_vectors /mnt/hgfs/binary-hack/lectures/3-debug-profile/leak.c:22:19
    #3 0x4c3241 in main /mnt/hgfs/binary-hack/lectures/3-debug-profile/leak.c:36:5
    #4 0x7f3f9c1bf0b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16

SUMMARY: AddressSanitizer: 80 byte(s) leaked in 1 allocation(s).
```

LLVM address sanitiser demonstration

- The Fibonacci buffer overflow is detected

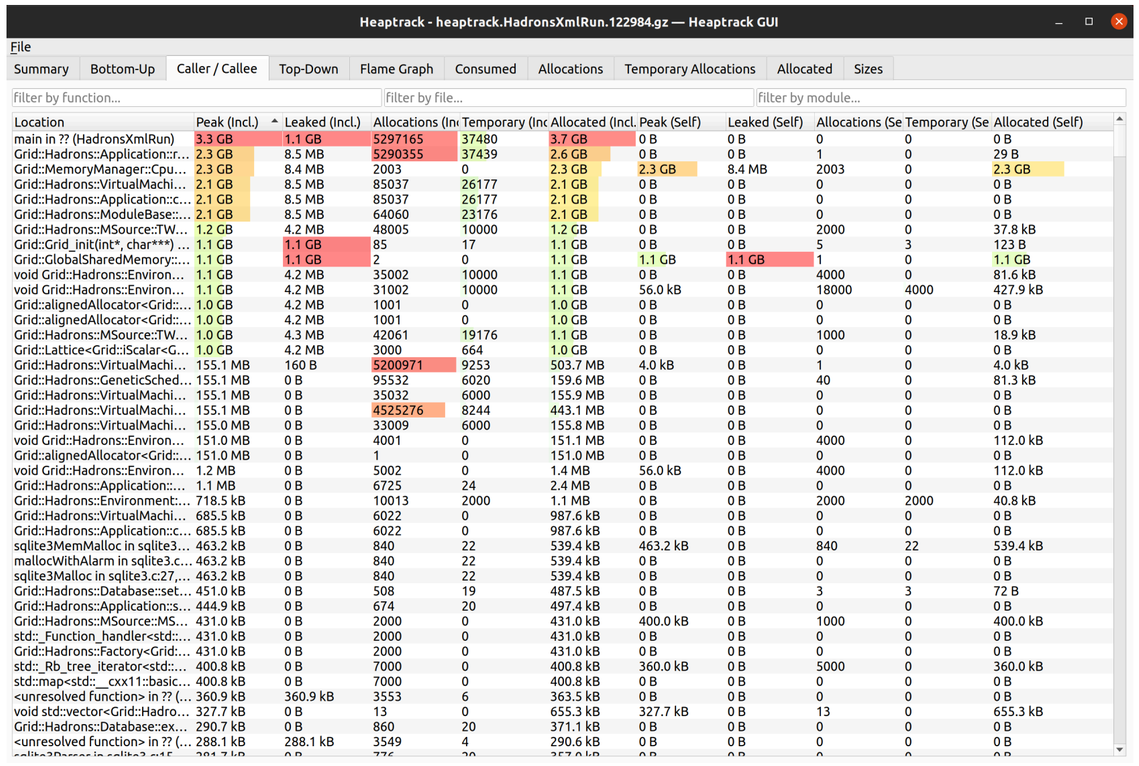
```
$ clang -fsanitize=address -fno-omit-frame-pointer -g boverflow.c -o boverflow
$ ./boverflow  aaaaaaaaaaa
Starting Fibonacci sequence with label aaaaaaaaaaa
1 1 2 3 5 8 13 21 34 55 89 144
$ ./boverflow  aaaaaaaaaaaaaaaaaa
=================================================================
==31124==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffc0fa004f0 at pc 0x00000047fb5e bp 0x7ffc0fa004b0 sp 0x7ffc0f9ffc70
WRITE of size 19 at 0x7ffc0fa004f0 thread T0
    #0 0x47fb5d in strcpy (/mnt/hgfs/binary-hack/lectures/3-debug-profile/boverflow+0x47fb5d)
    #1 0x4c324c in main /mnt/hgfs/binary-hack/lectures/3-debug-profile/boverflow.c:10:5
    #2 0x7fc7290ae0b2 in __libc_start_main /build/glibc-eX1tMB/glibc-2.31/csu/../csu/libc-start.c:308:16
    #3 0x41b2dd in _start (/mnt/hgfs/binary-hack/lectures/3-debug-profile/boverflow+0x41b2dd)

Address 0x7ffc0fa004f0 is located in stack of thread T0 at offset 48 in frame
    #0 0x4c311f in main /mnt/hgfs/binary-hack/lectures/3-debug-profile/boverflow.c:6

... then a lot of details about the overflow structure
... run it yourself and try to understand it!
```

Valgrind & heaptrack

- Valgrind is an excellent memory debugger, mainly aimed at memory leak detection

```
==43519== 80 bytes in 1 blocks are definitely lost in loss record 1 of 1
==43519==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==43519==    by 0x40116B: make_vector (leak.c:7)
==43519==    by 0x4011FC: swap_vectors (leak.c:22)
==43519==    by 0x401281: main (leak.c:36)
```

- heaptrack can track all allocations with minimal performance impact, it also has a GUI



Location	Peak (Incl.)	Leaked (Incl.)	Allocations (In	Temporary (Inc	Allocated (Incl.	Peak (Self)	Leaked (Self)	Allocations (Se	Temporary (Se	Allocated (Self)
main in ?? (HadronsXmlRun)	3.3 GB	1.1 GB	5297165	37480	3.7 GB	0 B	0 B	0	0	0 B
Grid::Hadrons::Application::r…	2.3 GB	8.5 MB	5290355	37439	2.6 GB	0 B	0 B	1	0	29 B
Grid::MemoryManager::Cpu…	2.3 GB	8.4 MB	2003	0	2.3 GB	2.3 GB	8.4 MB	2003	0	2.3 GB
Grid::Hadrons::VirtualMachi…	2.1 GB	8.5 MB	85037	26177	2.1 GB	0 B	0 B	0	0	0 B
Grid::Hadrons::Application::c…	2.1 GB	8.5 MB	85037	26177	2.1 GB	0 B	0 B	0	0	0 B
Grid::Hadrons::ModuleBase::…	2.1 GB	8.5 MB	64060	23176	2.1 GB	0 B	0 B	0	0	0 B
Grid::Hadrons::MSource::TW…	1.2 GB	4.2 MB	48005	10000	1.2 GB	0 B	0 B	2000	0	37.8 kB
Grid::Grid_init(int*, char***) …	1.1 GB	1.1 GB	85	17	1.1 GB	0 B	0 B	5	3	123 B
Grid::GlobalSharedMemory::…	1.1 GB	1.1 GB	2	0	1.1 GB	1.1 GB	1.1 GB	1	0	1.1 GB
void Grid::Hadrons::Environ…	1.1 GB	4.2 MB	35002	10000	1.1 GB	0 B	0 B	4000	0	81.6 kB
void Grid::Hadrons::Environ…	1.1 GB	4.2 MB	31002	10000	1.1 GB	56.0 kB	0 B	18000	4000	427.9 kB
Grid::alignedAllocator<Grid::…	1.0 GB	4.2 MB	1001	0	1.0 GB	0 B	0 B	0	0	0 B
Grid::alignedAllocator<Grid::…	1.0 GB	4.2 MB	1001	0	1.0 GB	0 B	0 B	0	0	0 B
Grid::Hadrons::MSource::TW…	1.0 GB	4.3 MB	42061	19176	1.1 GB	0 B	0 B	1000	0	18.9 kB
Grid::Lattice<Grid::iScalar<G…	1.0 GB	4.2 MB	3000	664	1.0 GB	0 B	0 B	0	0	0 B
Grid::Hadrons::VirtualMachi…	155.1 MB	160 B	5200971	9253	503.7 MB	4.0 kB	0 B	1	0	4.0 kB
Grid::Hadrons::GeneticSched…	155.1 MB	0 B	95532	6020	159.6 MB	0 B	0 B	40	0	81.3 kB
Grid::Hadrons::VirtualMachi…	155.1 MB	0 B	35032	6000	155.9 MB	0 B	0 B	0	0	0 B
Grid::Hadrons::VirtualMachi…	155.1 MB	0 B	4525276	8244	443.1 MB	0 B	0 B	0	0	0 B
Grid::Hadrons::VirtualMachi…	155.0 MB	0 B	33009	6000	155.8 MB	0 B	0 B	0	0	0 B
void Grid::Hadrons::Environ…	151.0 MB	0 B	4001	0	151.1 MB	0 B	0 B	4000	0	112.0 kB
Grid::alignedAllocator<Grid::…	151.0 MB	0 B	1	0	151.0 MB	0 B	0 B	0	0	0 B
void Grid::Hadrons::Environ…	1.2 MB	0 B	5002	0	1.4 MB	56.0 kB	0 B	4000	0	112.0 kB
Grid::Hadrons::Application::…	1.1 MB	0 B	6725	24	2.4 MB	0 B	0 B	0	0	0 B
Grid::Hadrons::Environment::…	718.5 kB	0 B	10013	2000	1.1 MB	0 B	0 B	2000	2000	40.8 kB
Grid::Hadrons::VirtualMachi…	685.5 kB	0 B	6022	0	987.6 kB	0 B	0 B	0	0	0 B
Grid::Hadrons::Application::c…	685.5 kB	0 B	6022	0	987.6 kB	0 B	0 B	0	0	0 B
sqlite3MemMalloc in sqlite3…	463.2 kB	0 B	840	22	539.4 kB	463.2 kB	0 B	840	22	539.4 kB
mallocWithAlarm in sqlite3.c…	463.2 kB	0 B	840	22	539.4 kB	0 B	0 B	0	0	0 B
sqlite3Malloc in sqlite3.c:27,…	463.2 kB	0 B	840	22	539.4 kB	0 B	0 B	0	0	0 B
Grid::Hadrons::Database::set…	451.0 kB	0 B	508	19	487.5 kB	0 B	0 B	3	3	72 B
Grid::Hadrons::Application::s…	444.9 kB	0 B	674	20	497.4 kB	0 B	0 B	0	0	0 B
Grid::Hadrons::MSource::MS…	431.0 kB	0 B	2000	0	431.0 kB	400.0 kB	0 B	1000	0	400.0 kB
std::_Function_handler<std::…	431.0 kB	0 B	2000	0	431.0 kB	0 B	0 B	0	0	0 B
Grid::Hadrons::Factory<Grid::…	431.0 kB	0 B	2000	0	431.0 kB	0 B	0 B	0	0	0 B
std::_Rb_tree_iterator<std::…	400.8 kB	0 B	7000	0	400.8 kB	360.0 kB	0 B	5000	0	360.0 kB
std::map<std::__cxx11::basic…	400.8 kB	0 B	7000	0	400.8 kB	0 B	0 B	0	0	0 B
<unresolved function> in ?? (…	360.9 kB	360.9 kB	3553	6	363.5 kB	0 B	0 B	0	0	0 B
void std::vector<Grid::Hadro…	327.7 kB	0 B	13	0	655.3 kB	327.7 kB	0 B	13	0	655.3 kB
Grid::Hadrons::Database::ex…	290.7 kB	0 B	860	20	371.1 kB	0 B	0 B	0	0	0 B
<unresolved function> in ?? (…	288.1 kB	288.1 kB	3549	4	290.6 kB	0 B	0 B	0	0	0 B

Make your own!

- There are various methods for writing your own code instrumentations

- Cf. next lecture

Profiling and optimisation

What is profiling

- Profiling generally entails measuring metrics during a program execution (time, memory, …) to establish a profile of the heaviest routines

- This can be done just has a check that a program behaves normally, but also to guide further optimisations

- This can be done at the code level using **timers and print statements**

- This can be done through **code injection** (cf. next lecture) and **instrumentation**

- This can also be done using specialised tools, **profilers**

Profiling with gperftools

- Originally developed by Google, cross-platform (Unix) https://bit.ly/2O9dFrq
- Does CPU and memory profiling
- CPU example (source https://bit.ly/2ZY6p4e)

```
$ clang ./profileme.c -O3 -march=native -o profileme -g -lm -lprofiler
$ CPUPROFILE=prof CPUPROFILE_FREQUENCY=1000 ./profileme
PROFILE: interrupts/evictions/bytes = 612/487/23504
$ google-pprof --text ./profileme prof
Using local file ./profileme.
Using local file prof.
Total: 612 samples
     444  72.5%  72.5%      444  72.5% f64xsubf128
      76  12.4%  85.0%       76  12.4% acos
      47   7.7%  92.6%      356  58.2% vec_acos (inline)
      20   3.3%  95.9%      249  40.7% vec_sin (inline)
      17   2.8%  98.7%       17   2.8% _init
       7   1.1%  99.8%        7   1.1% vec_acc (inline)
       1   0.2% 100.0%        1   0.2% GLIBC_2.15
       0   0.0% 100.0%      612 100.0% __libc_start_main
       0   0.0% 100.0%      612 100.0% _start
       0   0.0% 100.0%      612 100.0% main
```

Other CPU profilers

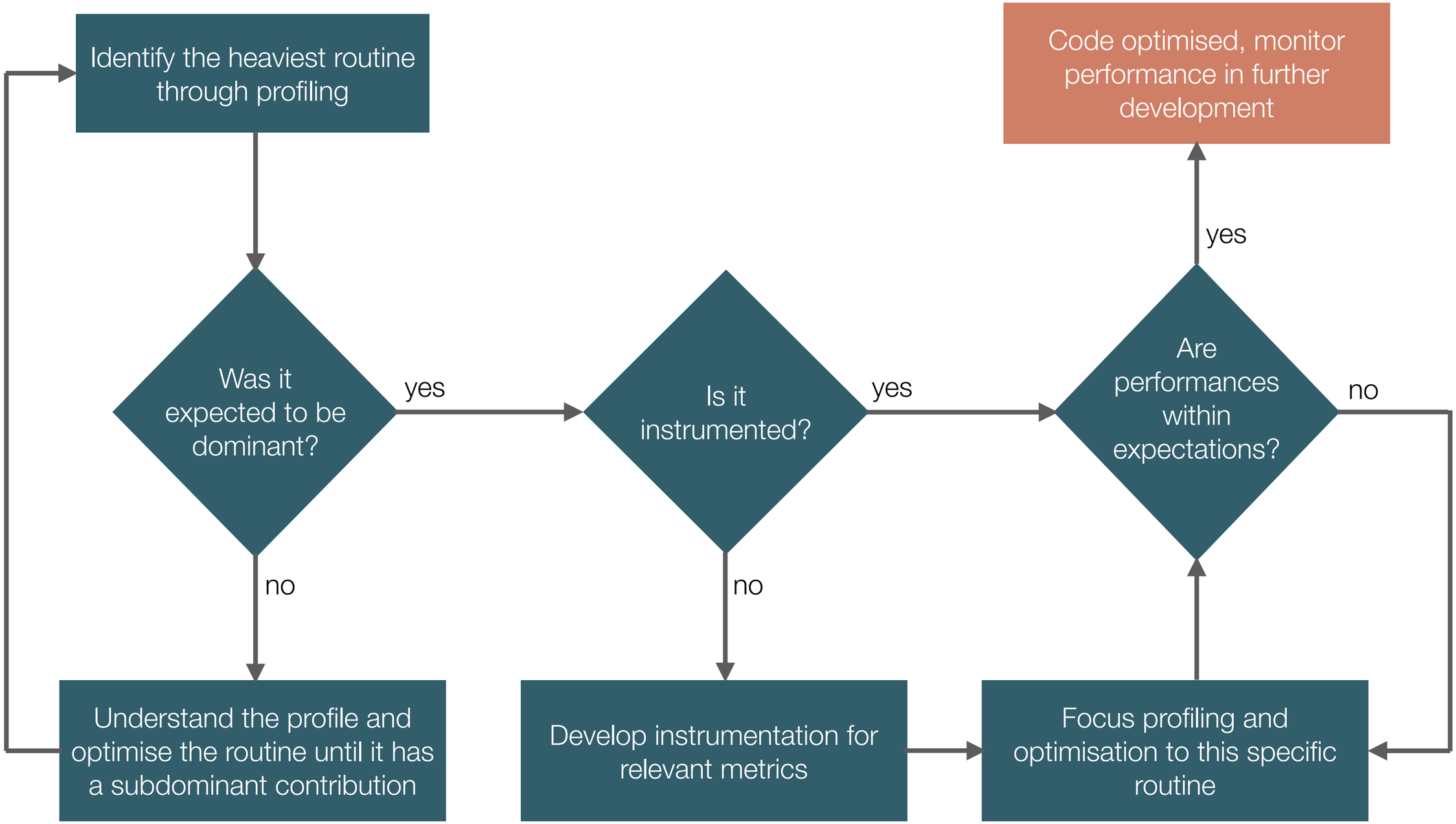
- Linux kernel own profiler https://bit.ly/3uGUk1m
  Very deep, based on hardware counters.
  May need root to change kernel settings

- Intel VTune https://intel.ly/3q1VGQM
  Proprietary, only for Intel processors
  Likely to be installed on Intel-based clusters

- AMD μProf https://bit.ly/3r1Zx1v
  Similar to VTune for AMD processors

- Apple Instruments https://apple.co/3q1EsDd
  Very neat GUI to browse traces, macOS only…

Optimisation workflow



It depends a lot on building **prior expectations**…

The roofline model

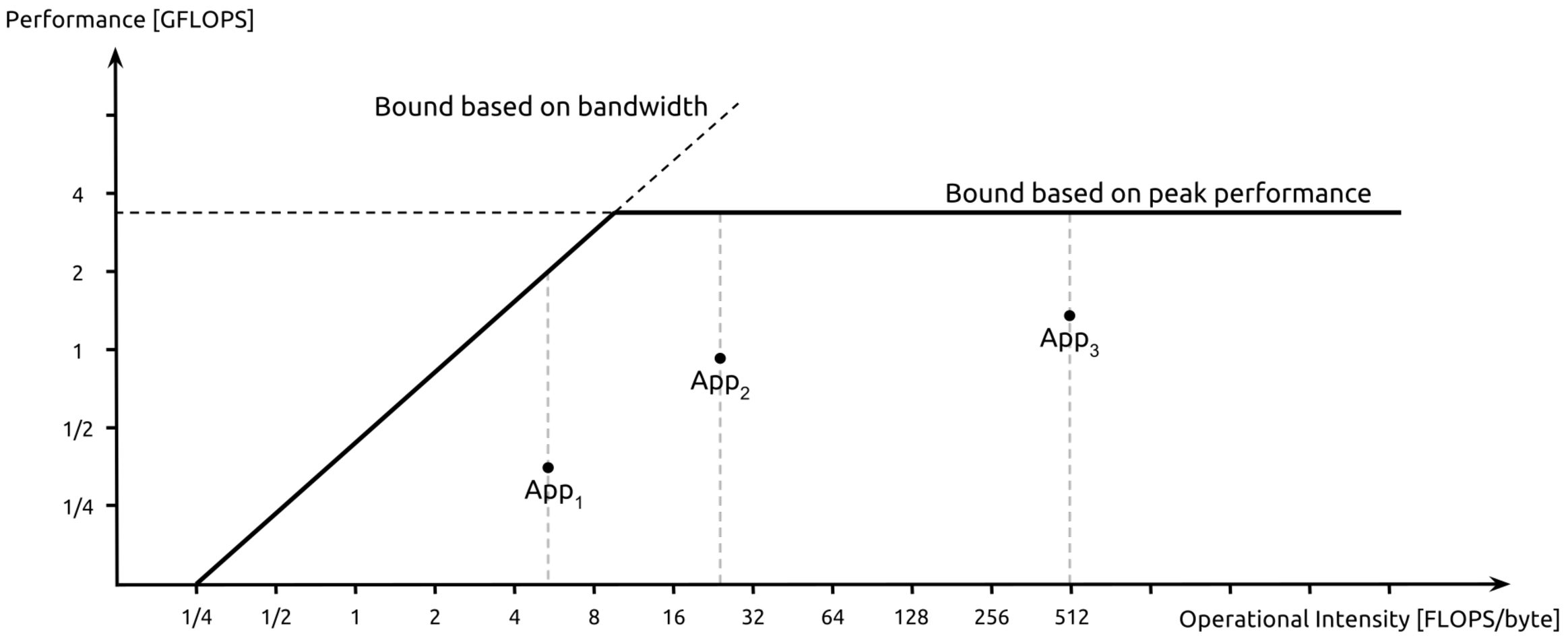
- The roofline model is a **simple performance model** for assessing the optimisation of applications **dominated by floating-point (FP) arithmetic**

- To first approximation, a FP calculation **read numbers** and **performs operations** on them

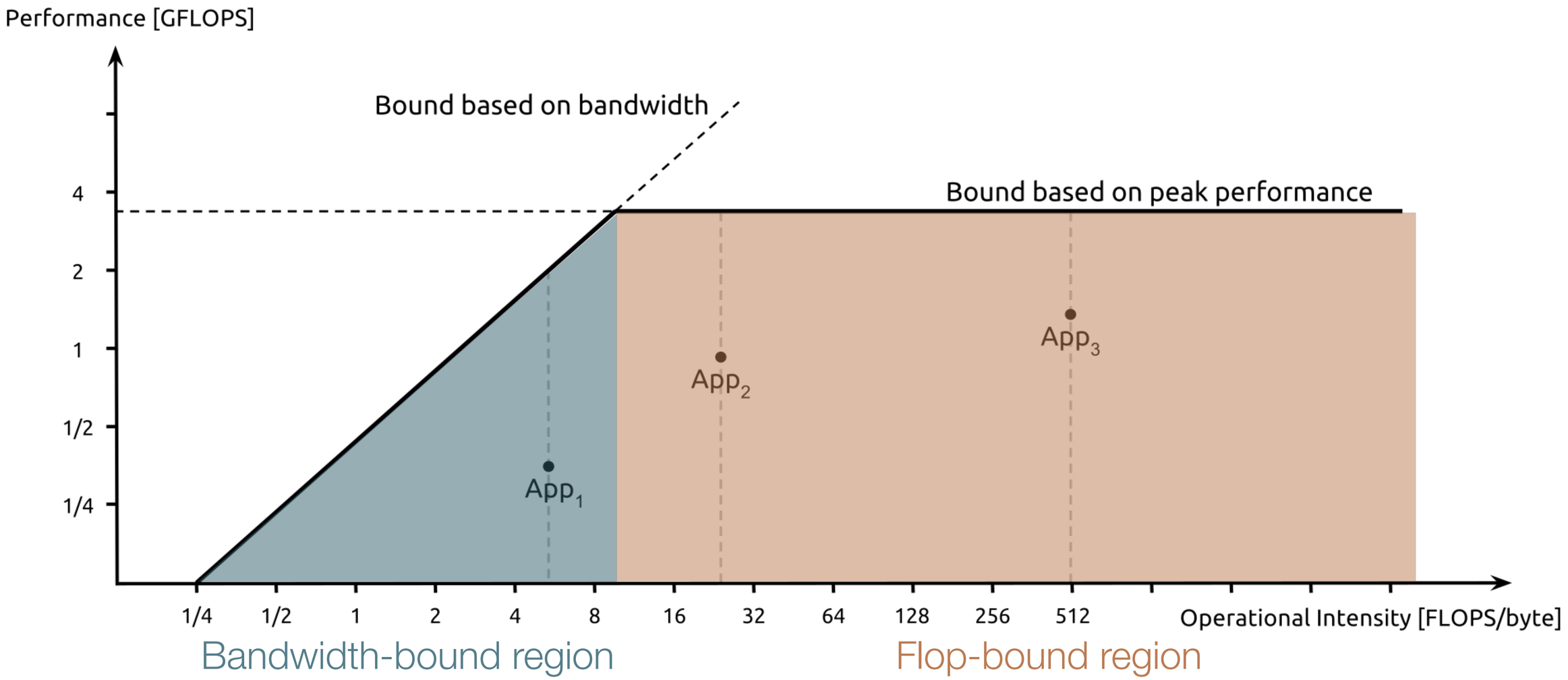
- The roofline model describe the limit of a computer using **2 dimensions**:
  1) the maximum FP operations (Flop) per second
  2) the maximum memory bandwidth in byte per second

- A code is considered optimal if it hits one of these bounds

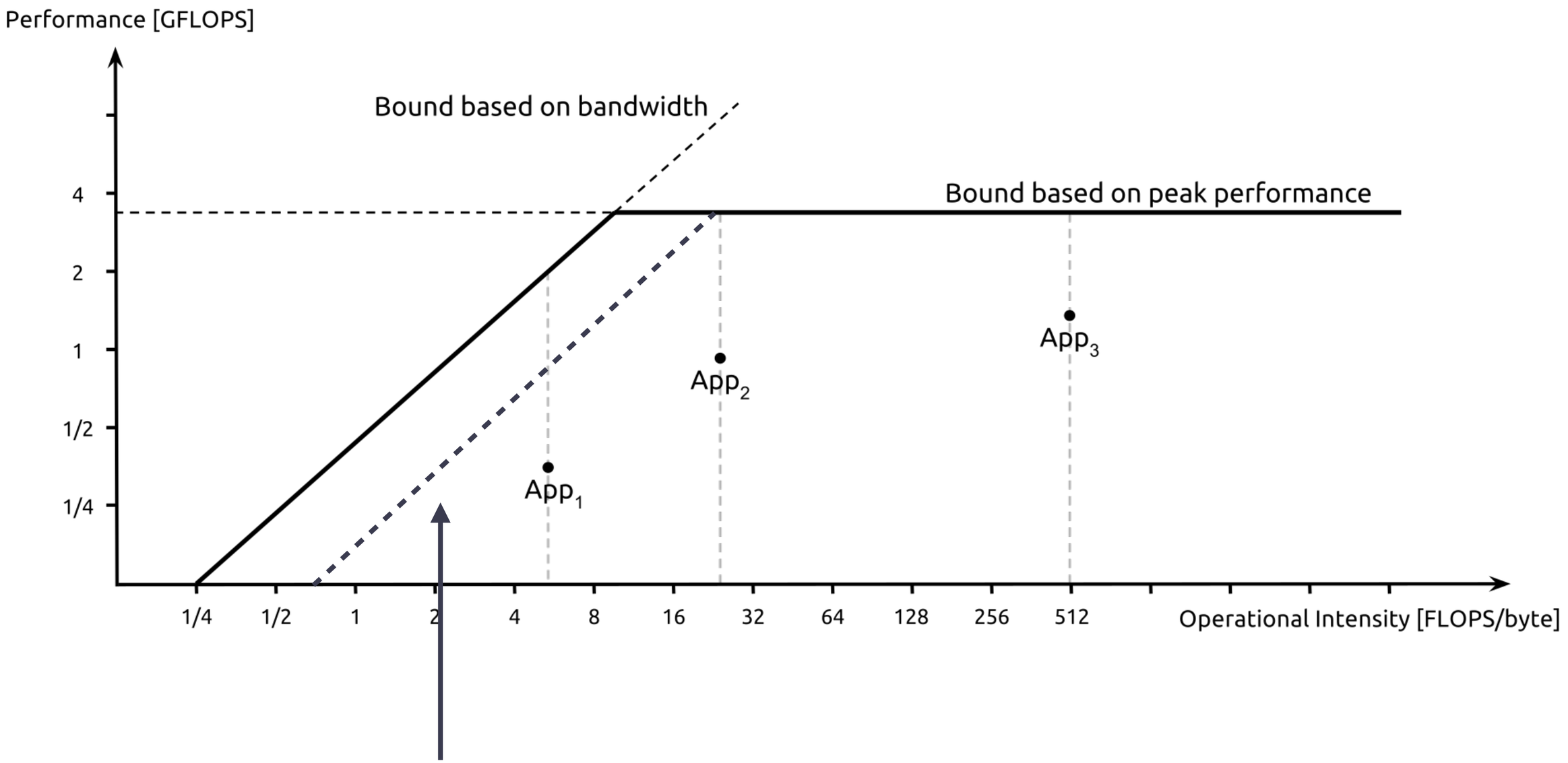
The roofline model



The roofline model



The roofline model



New bounds & regions can be added
(e.g. network bandwidth for parallel applications, cache memory)

Theoretical Flop/s peaks

- For NVIDIA GPUs: easy, provided in GPU spec-sheet

- For x86 CPU: not always provided by vendors

$$R_{\text{peak}} = n_{\text{cores}} \times f \times c$$

n_{cores} : number of physical cores

f : max (sometime called “turbo”) clock frequency

c : max FP instruction per clock cycle

- c is non-trivial, it depends on the ISA and the number of FP units on the chip
  Really useful table: https://bit.ly/2ZWpELH
  Intel data: https://intel.ly/3qVBpxt

daxpy: example of bandwidth-bound routine

- "daxpy" is the classic BLAS level 1 routine

$$\mathbf{y} \leftarrow a\mathbf{x} + \mathbf{y}$$

  $\mathbf{x}$ & $\mathbf{y}$ double-precision FP vectors of size n

- Bytes read/write (double precision): $8(3n + 1)$ (STREAM convention)

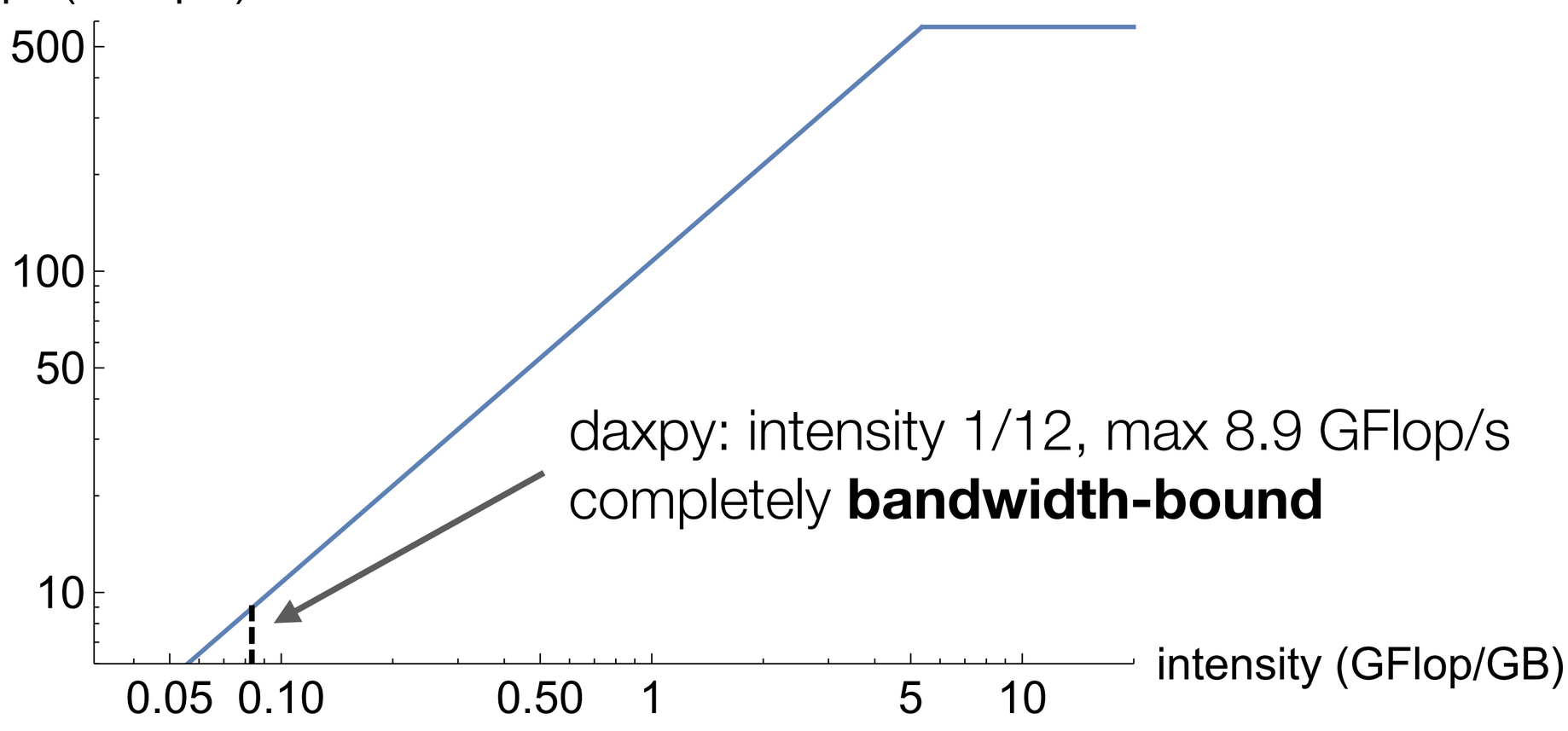
- FP operations: $2n$

- Operational intensity (large n): $1/12$

- **Low-intensity**: very likely **bandwidth-bound**

Practical case: Intel Xeon Silver 4116

- Peaks: 576 GFlop/s (double)— 107.3 GB/s
- References: https://intel.ly/2ZLjrSG
  https://bit.ly/3aOVTTf



Xeon Silver 4116 roofline model

daxpy implementations

- "Naive" C loop

```
1 void axpy_ptr(const double a, const double *x, double *y, const size_t size)
2 {
3     for (unsigned int i = 0; i < size; ++i)
4     {
5         y[i] = a*x[i] + y[i];
6     }
7 }
```

- Threaded C loop

```
1 void axpy_ptr_thread(const double a, const double *x, double *y, const size_t size)
2 {
3     #pragma omp parallel for
4     for (unsigned int i = 0; i < size; ++i)
5     {
6         y[i] = a*x[i] + y[i];
7     }
8 }
```

daxpy implementations

- High-level C++ version (can be threaded too)

```
1 void axpy_vec(const double a, const vector<double> &x, vector<double> &y)
2 {
3     for (unsigned int i = 0; i < x.size(); ++i)
4     {
5         y[i] = a*x[i] + y[i];
6     }
7 }
```

- Highly optimised Intel MKL 2019 (automatically threaded)

```
1 void axpy_mkl(const double a, const double *x, double *y, const size_t size)
2 {
3     cblas_daxpy(size, a, x, 1, y, 1);
4 }
```

Inputs and compilation

- Heap and stack arrays filled with random data

```
1 double xar[NARRAY], yar[NARRAY]; // will be stored on the stack
2                                  // NARRAY known at compile time
3 std::vector<double> x(vecSize), y(vecSize); // C++ vectors are contiguous heap buffers
4                                             // vecSize can only be known at run time
```

- Compilation

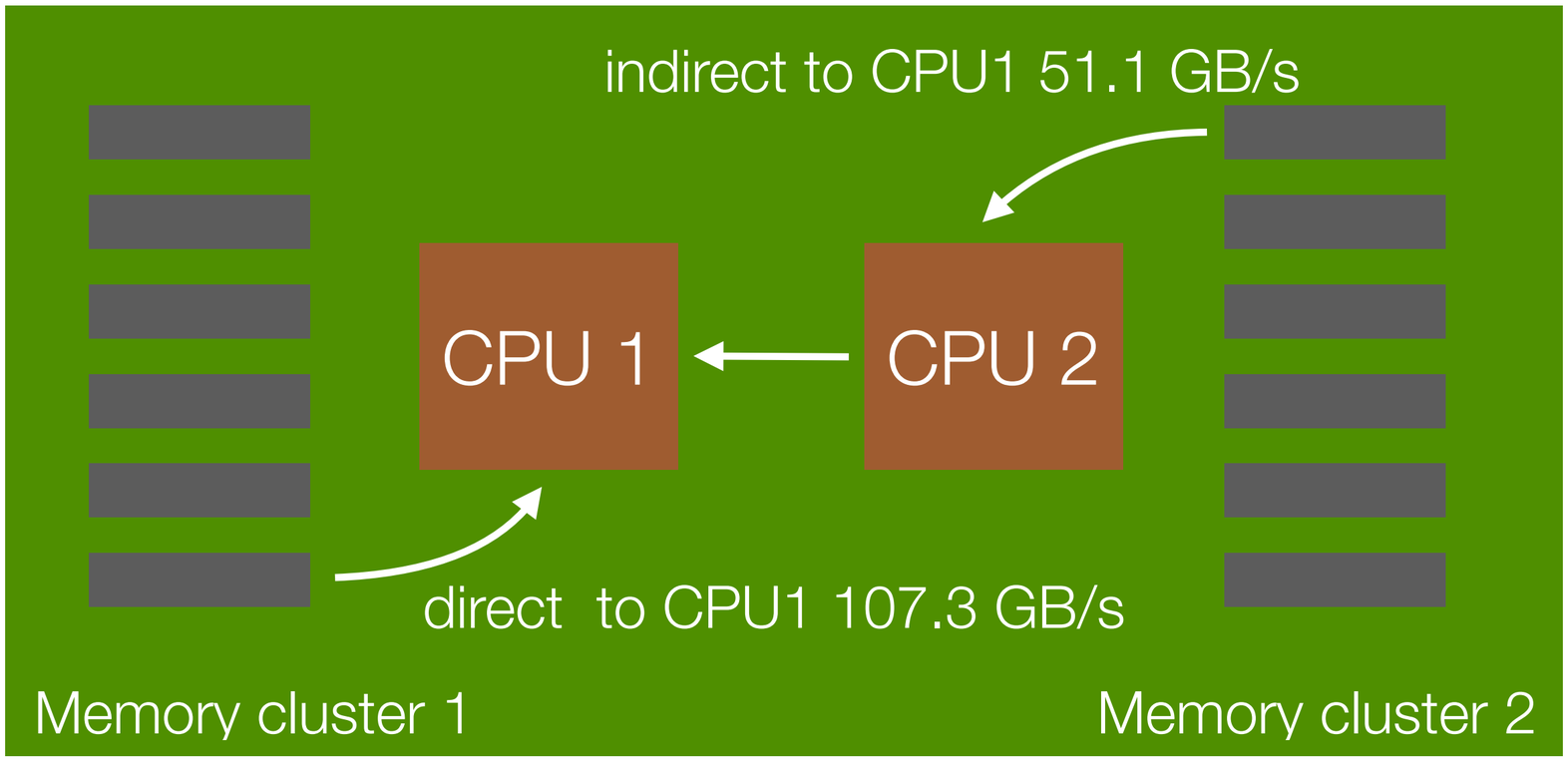
```
clang++ -g -fopenmp axpy.cpp -O3 -o axpy -march=skylake-avx512 -lmkl_rt
```

Notice **level 3 optimisations and AVX512 ISA**

- Full code

Running

- Subtlety: I am using a double socket node

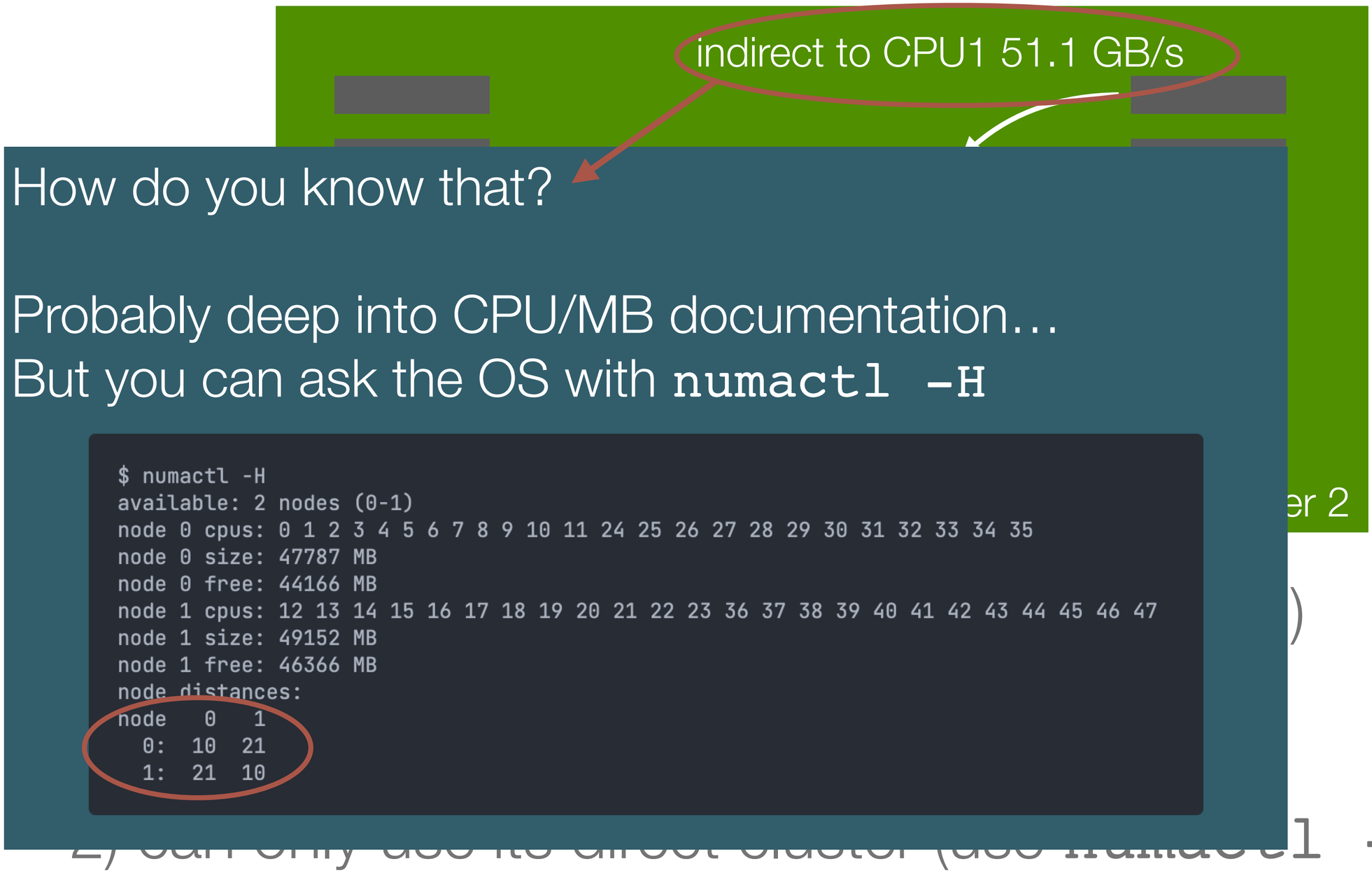


- To benchmark a **single CPU** (let's say CPU 1) we need to be sure that the process
  1) is confined to CPU 1 (OpenMP options)
  2) can only use its direct cluster (use `numactl -m`)

```
OMP_PLACES='cores(12)' OMP_NUM_THREADS=12 OMP_PROC_BIND=close numactl -m 0 ./axpy
```

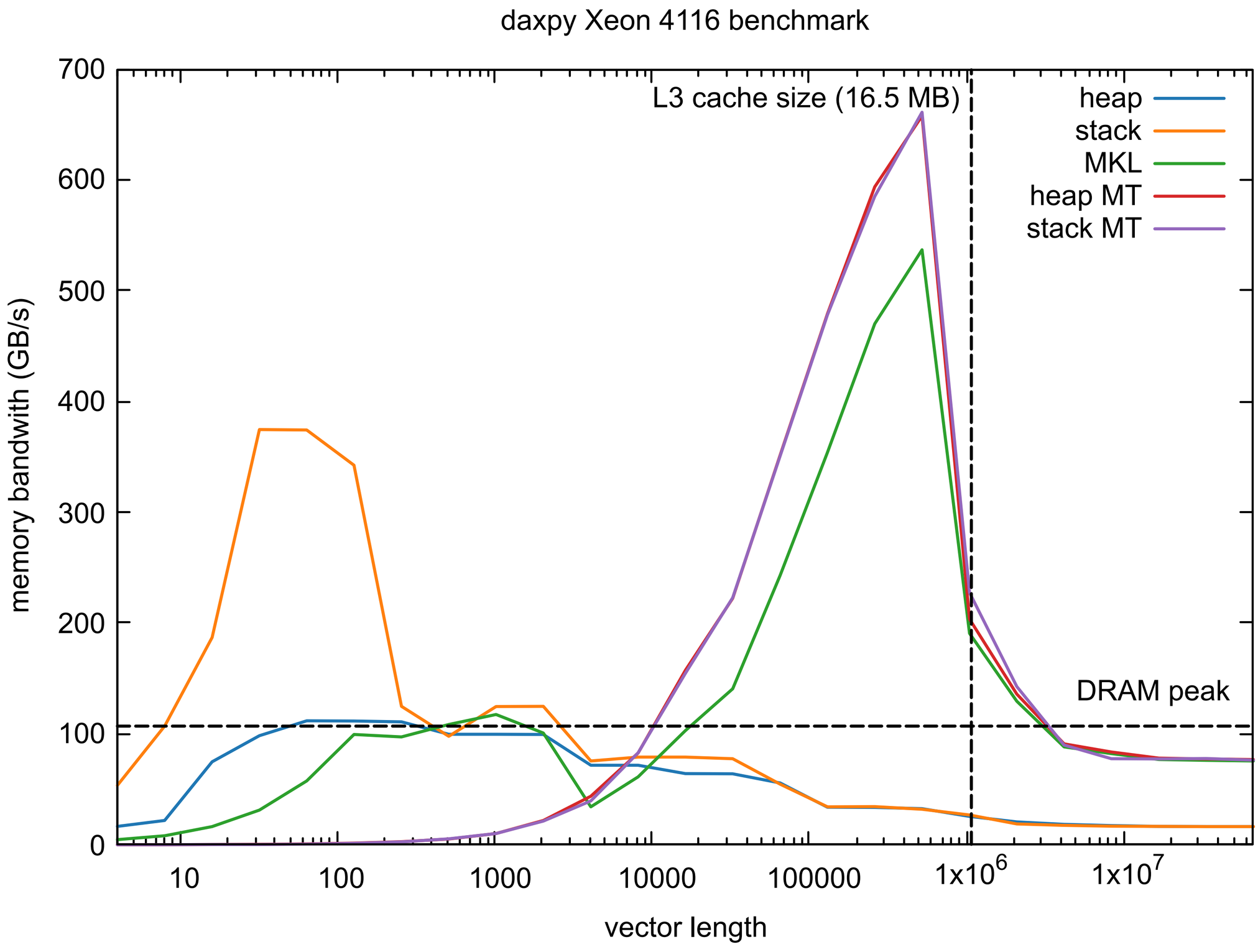
Running

- Subtlety: I am using a double socket node

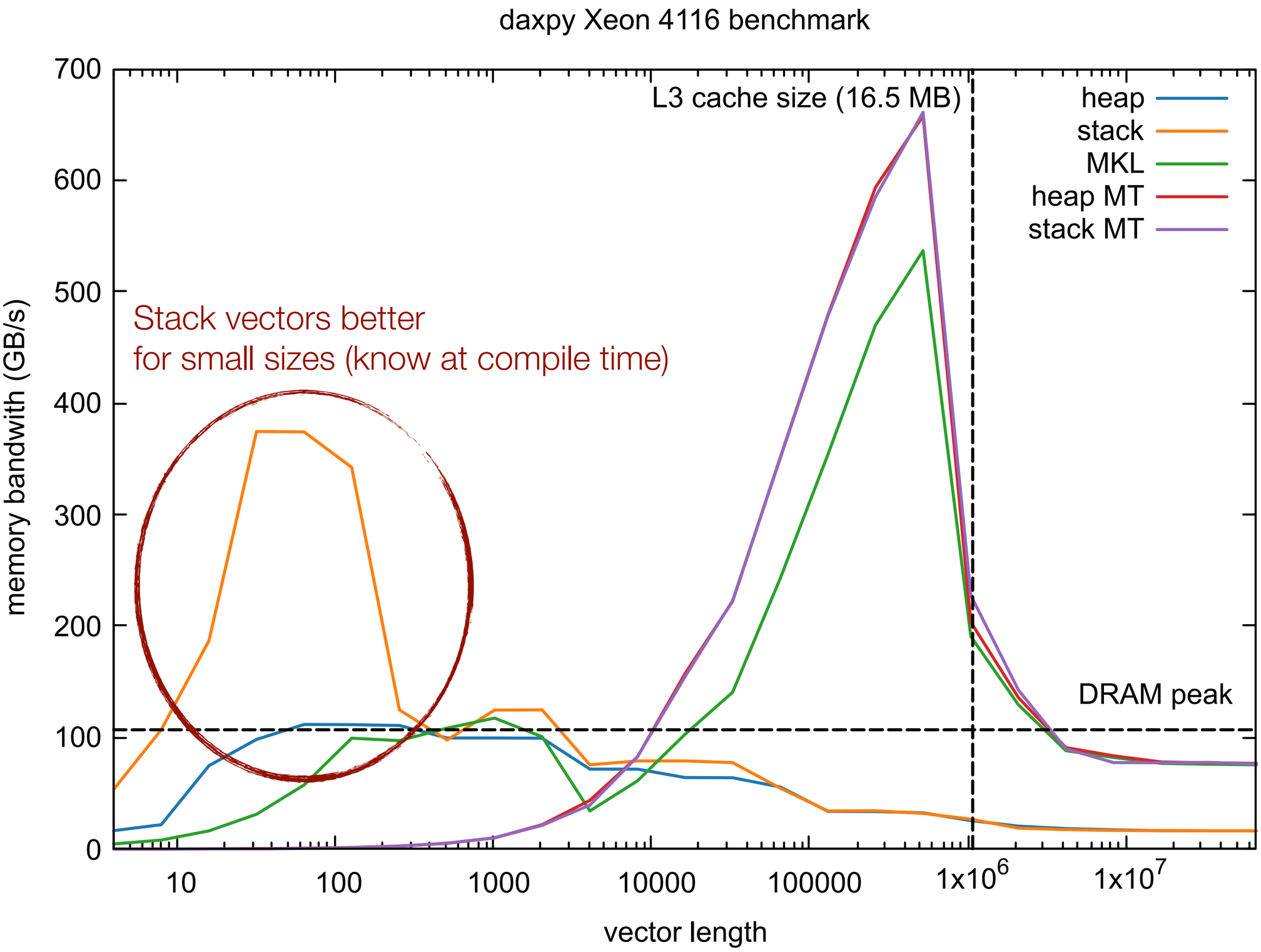



```
OMP_PLACES='cores(12)' OMP_NUM_THREADS=12 OMP_PROC_BIND=close numactl -m 0 ./axpy
```

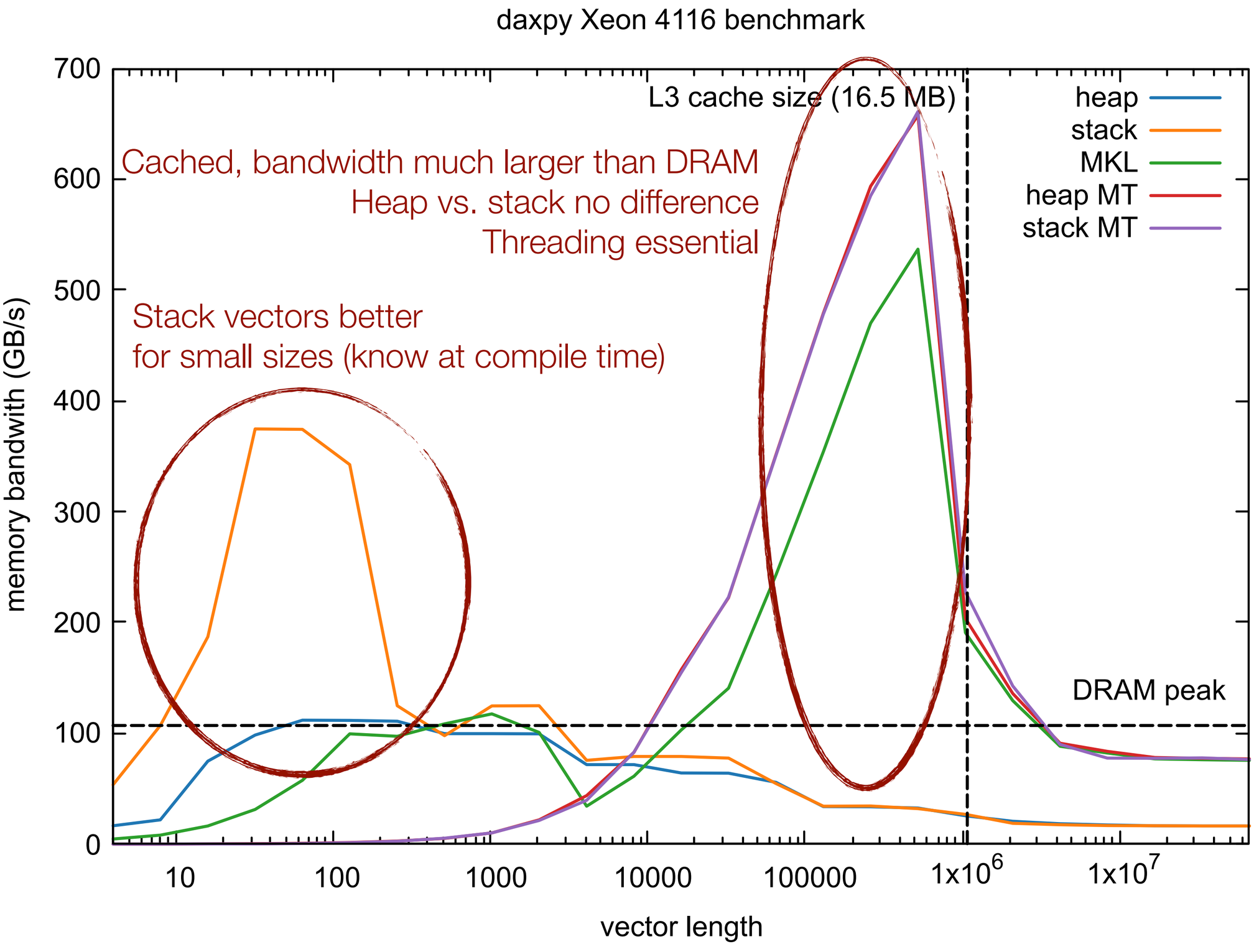
Benchmark results



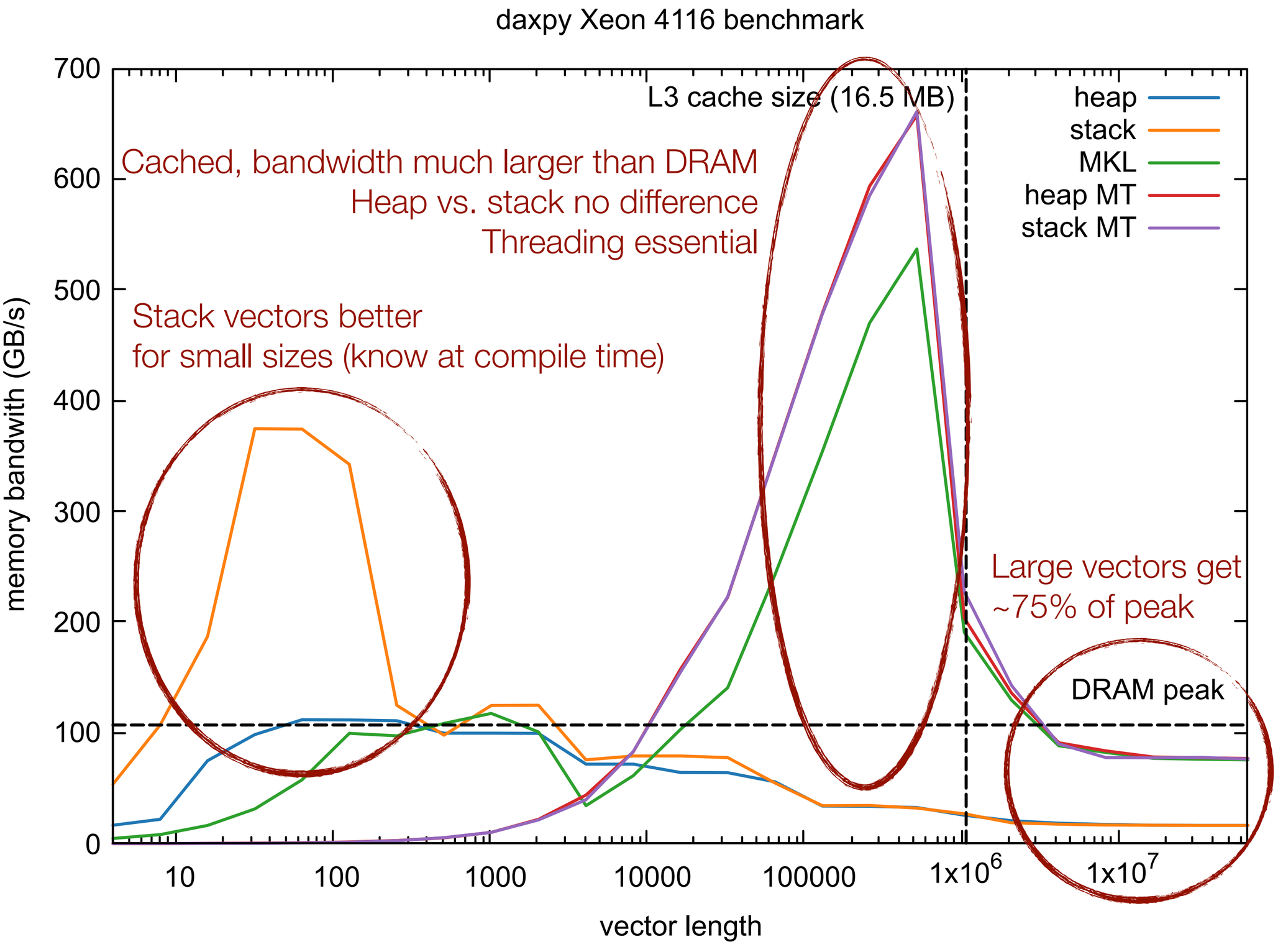
Benchmark results



Benchmark results



Benchmark results



daxpy conclusions

- **Naive code can be optimal** assuming
  1) maximum compiler optimisations
  2) good awareness of the hardware context

- Roofline model **really useful**

- **No silver bullet**, different optimal strategies for different sizes

- Why not 100%?

  1. Actually just counting read+write is somewhat naive, often CPUs read into the cache before writing
     Do the math… (1 more read), it brings us much closer

  2. Theoretical peaks are **ideal numbers** where the CPU purely read/write data or compute

dgemm: example of compute-bound problem

- "dgemm" level 3 BLAS routine

$$C \leftarrow \alpha AB + \beta C$$

$A,\ B$ & C FP double-precision matrices

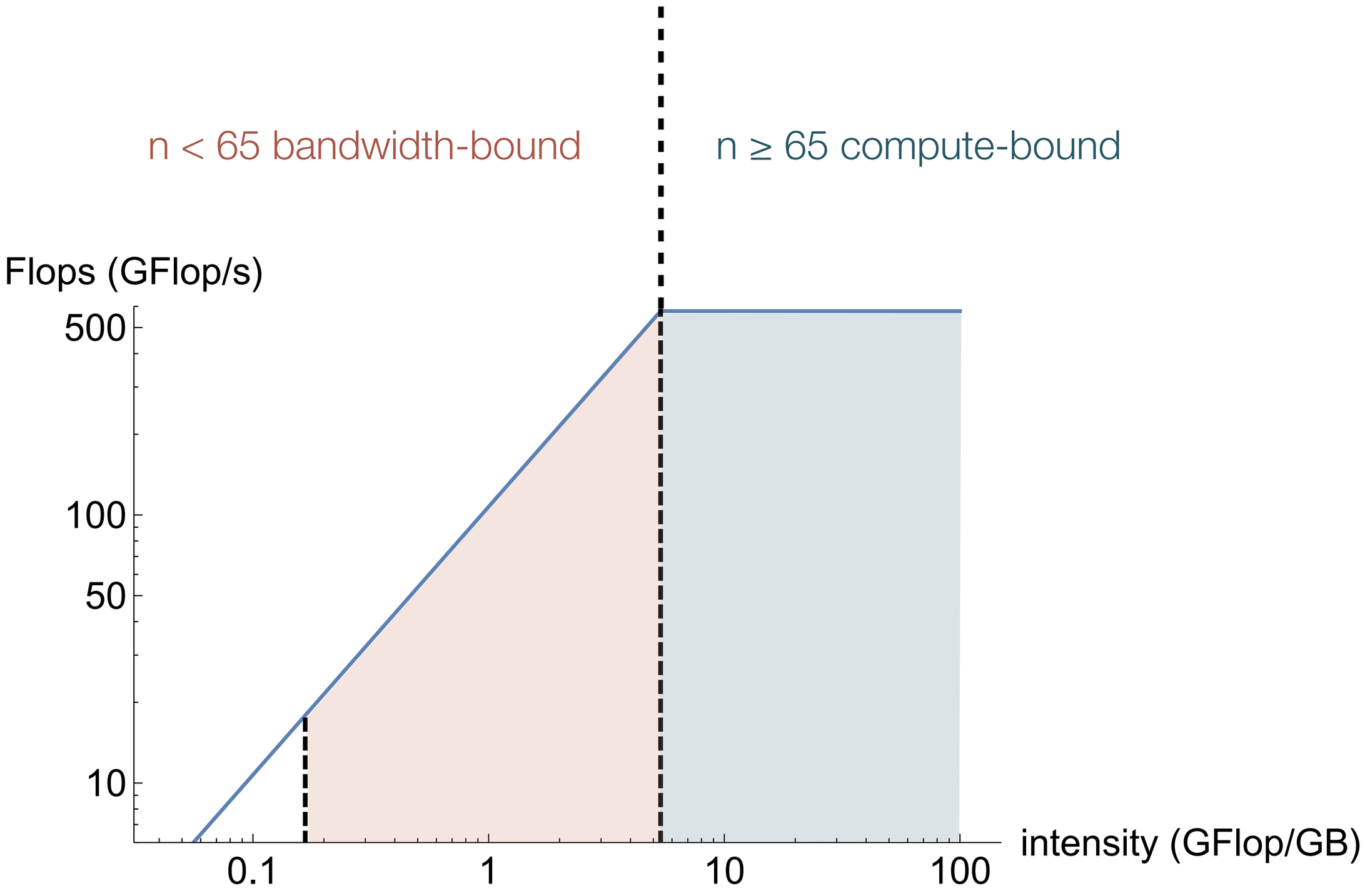
- Let's keep it simple: square $n \times n$ matrices, $\beta = 0$.
  So just square matrix product.

- Bytes read/write: $24n^2$

- FP operations: $2n^3$

- Intensity $n/12$, linearly growing with matrix size

Xeon 4116 dgemm roofline model



dgemm implementations

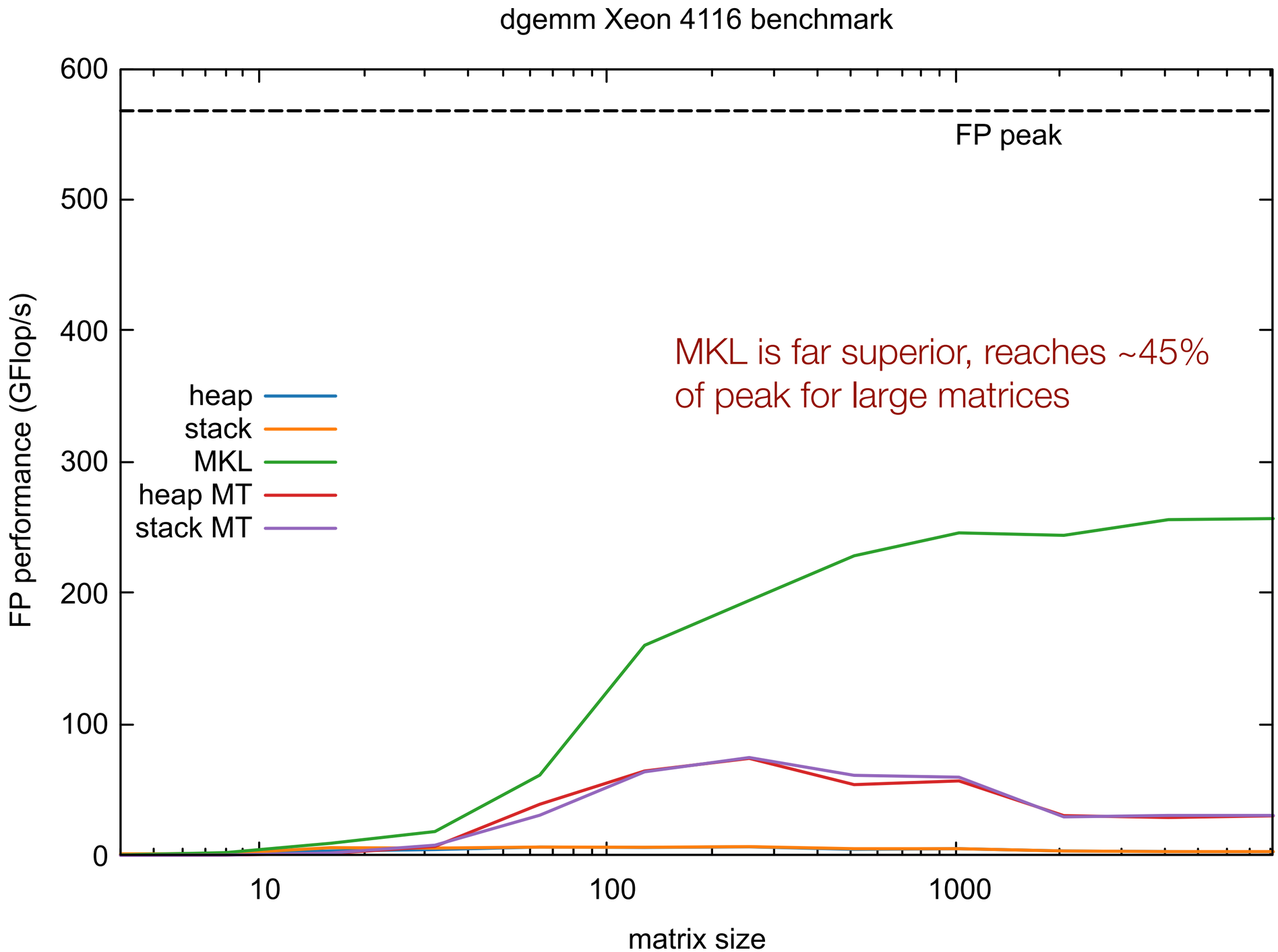
- "Naive" C loops

```
void dgemm_ptr(double *c, const double *a, const double *b, const size_t n)
{
    for (unsigned int i = 0; i < n; ++i)
    for (unsigned int j = 0; j < n; ++j)
    {
        c[i*n + j] = a[i*n]*b[j];
    }
    for (unsigned int i = 0; i < n; ++i)
    for (unsigned int k = 1; k < n; ++k)
    for (unsigned int j = 0; j < n; ++j)
    {
        c[i*n + j] += a[i*n + k]*b[k*n + j];
    }
}
```

- Intel MKL call

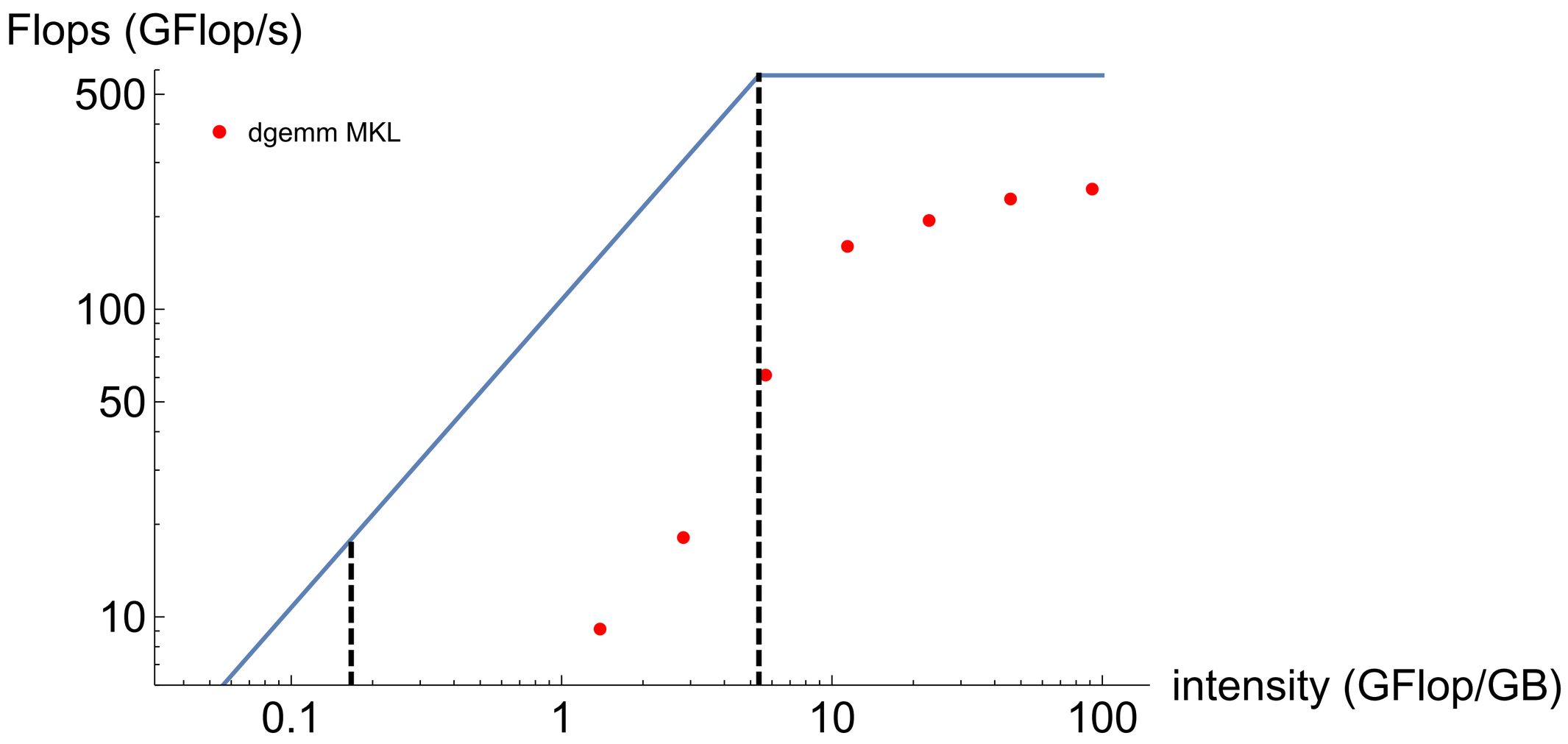
```
void dgemm_mkl(double *c, const double *a, const double *b, const size_t n)
{
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, n,
                n, 1., a, n, b, n, 0., c, n);
}
```

Benchmark results



Benchmark results

- Roofline view of the MKL dgemm benchmark



dgemm conclusions

- dgemm is **much less trivial than daxpy**, implementing dgemm is almost a field of research on its own…

- This is mainly due to the **non-trivial memory access pattern**, making caching and vectorisation difficult

- But clearly as expected **the memory bandwidth is not a constraint** in any cases

- Beyond MKL, a number of other implementations

  OpenBLAS, Eigen, Blitz, …

Final comments

- Theoretical peaks are **impossible to match in practice**, they assume the CPU is doing nothing else than accessing memory of performing FP operations

- Many effects will **take you away from the peaks**, dependencies, non-sequential access, impossibility to vectorise perfectly a formula, integer arithmetics necessary for loops, etc…

- For HPC: **do not abstract the hardware**! Be aware of it, its limitations and specifications

- Learn and experiment with effects of **hierarchical memory** (i.e. cache memory)