ETH zürich

Mixed-Data-Model Heterogeneous Compilation and OpenMP Offloading

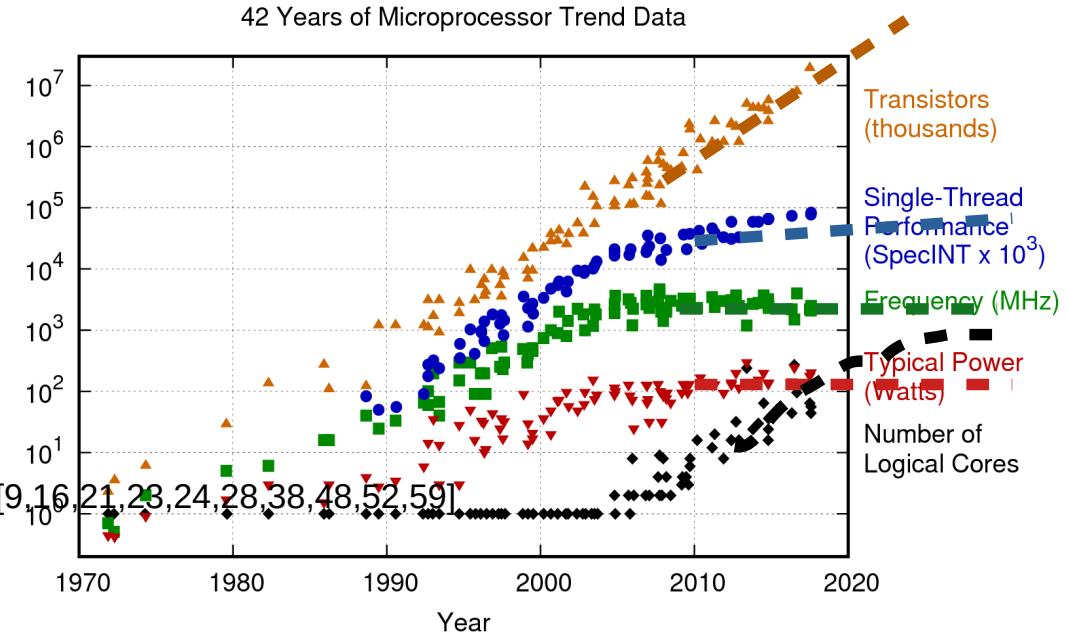
February 22, 2020
ACM SIGPLAN 2020 International Conference on Compiler Construction (CC 2020)

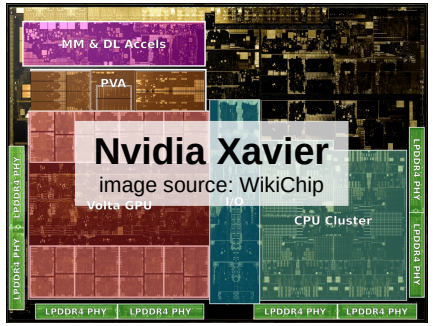
Andreas Kurth, Koen Wolters, Björn Forsberg, Alessandro Capotondi, Andrea Marongiu, Tobias Grosser, and Luca Benini

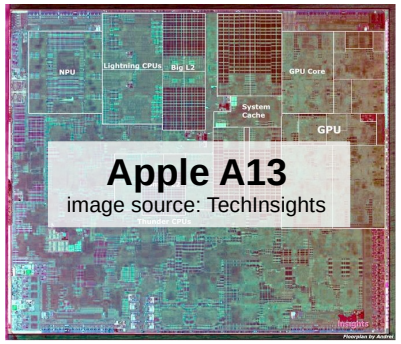
Integrated Systems Laboratory
Department of Electrical Engineering and Information Technology

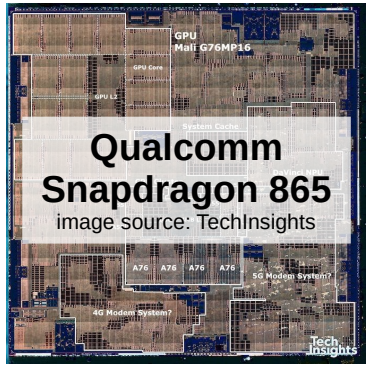
Why Heterogeneous Computing?

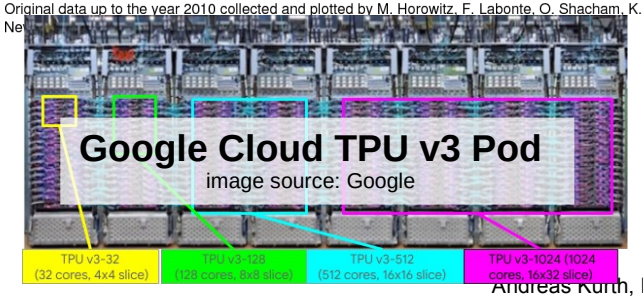
- **Paradigms** dominating **computer performance growth**:
  - 1970 – 2005: single-core IPC, SMT *until power wall*
  - 2005 – 202*?*: multi- to many-core *until diminishing returns (Amdahl's)*
  - 2015 – 20*??*: domain-specific accelerators (e.g., DSPs [5,10,17,37,51-54] and DSP-like accelerators [13,43], GPUs [1,2,4,39-41,45]) in **heterogeneous computers** [9,16,21,23,24,28,38,48,52,59]





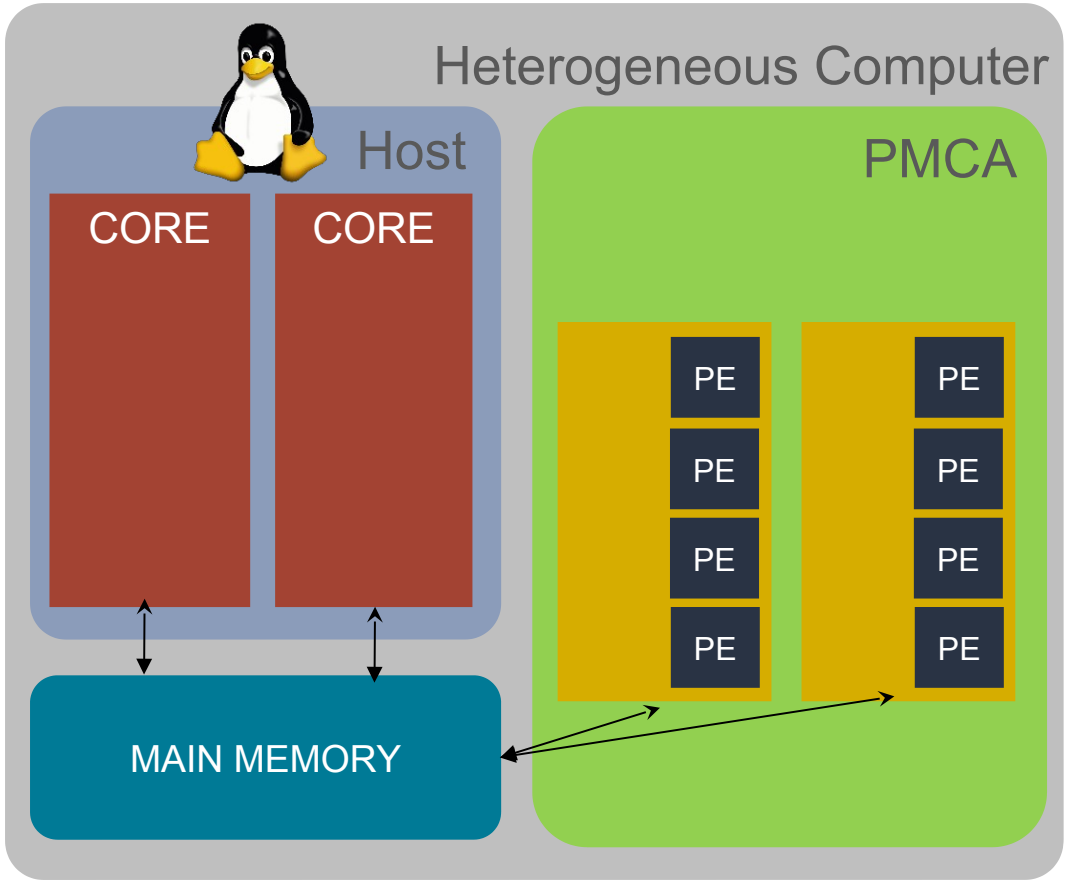






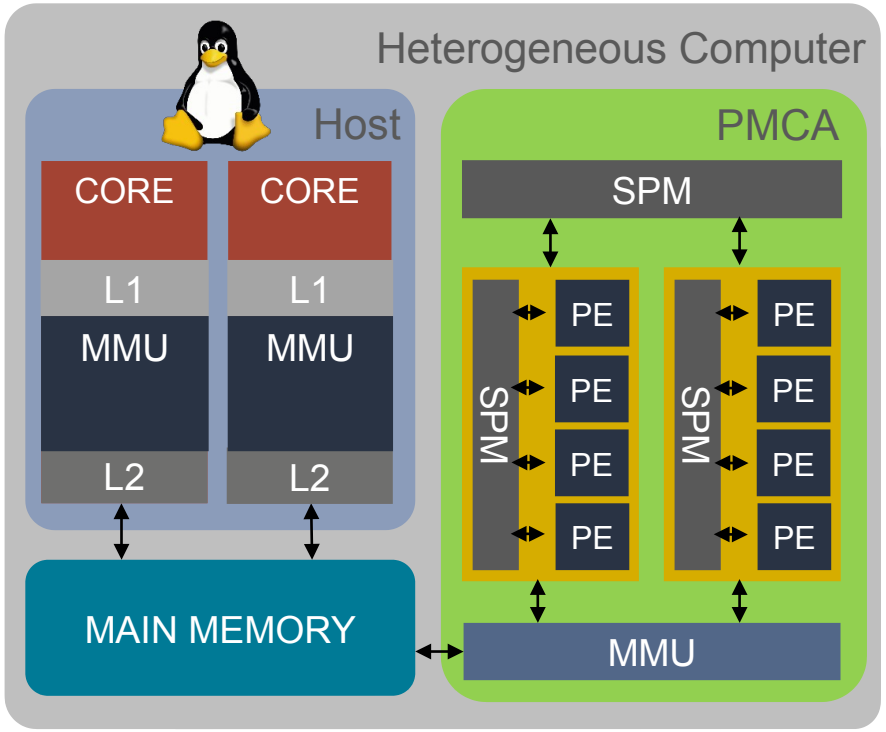
Heterogeneous Computer Architectures

- Goal: Combine
  - **general-purpose Host** CPU
    for versatility (e.g., standard OS, I/Os)
  - domain-specific **programmable many-core accelerators (PMCA)**
    for efficiency and performance
- Offload tasks from host to PMCA and **share data via main memory**.



Challenges of Heterogenous Computers

- Heterogeneous computers are **complex**: **Host** and **PMCAs** usually **differ** in
  - **instruction set architecture** (ISA) and/or **data model** (e.g., ILP32, LP64)
  - **memory hierarchy**: caches vs. scratchpad memories (SPMs)
  - **memory accessing**: cache line fetching vs. DMA bursts
- **Goal: Abstract complexity from users**
- **Transparent application offloading**:
  - Shared Virtual Memory (SVM [29,33,58])
  - Offloading API (e.g., OpenMP [44], OpenACC [7], OpenCL [47])

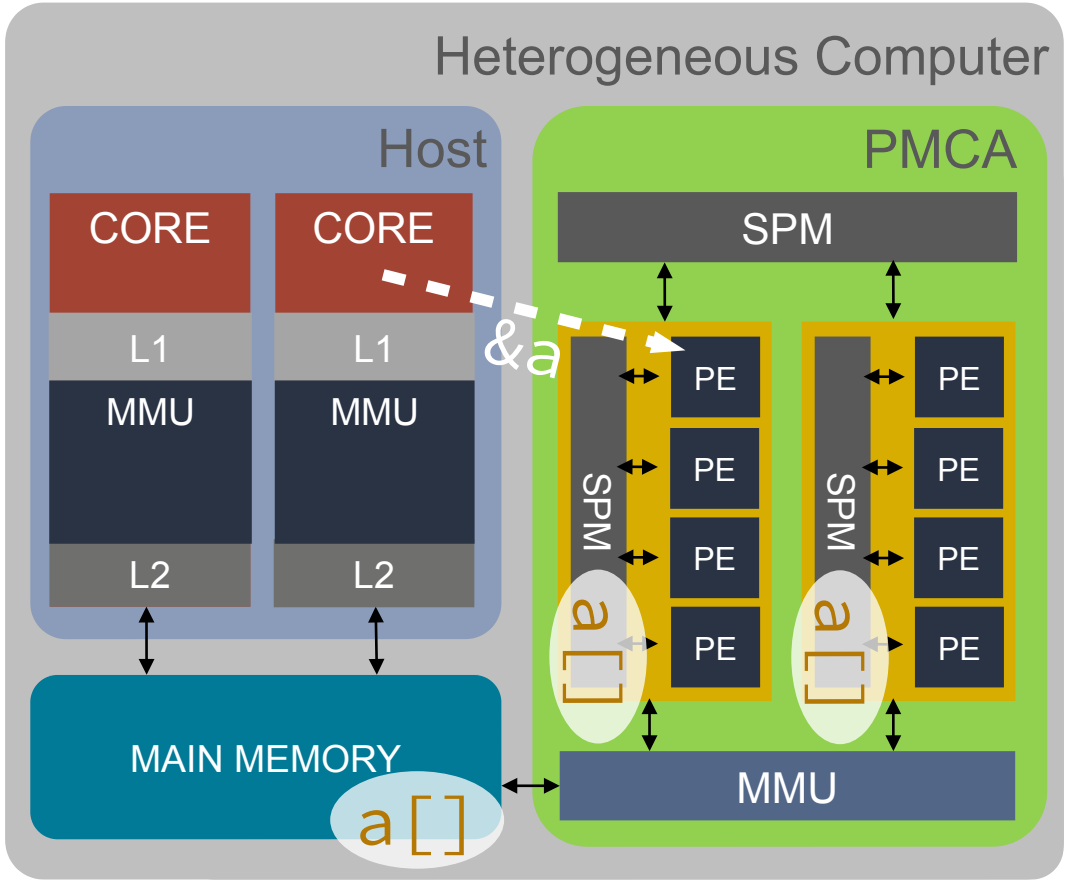



```
void host_function(int n, int* a) {
  #pragma omp target map(tofrom: a[0:n], n)
  {
    accelerator_function(n, a);
  }
}
```

Application Execution on Heterogeneous Computers

0. Data (e.g. a[]) resides in main memory.
1. At offload, Host passes pointer to arguments to PMCA.
2. PMCA ..
   1. **transfers** required data **to** its **local SPMs**
   2. **executes** function on data in local SPMs
   3. **transfers** data **back** as required
   4. potentially **repeats 1–3** (tiling)
   5. **signals completion** to Host

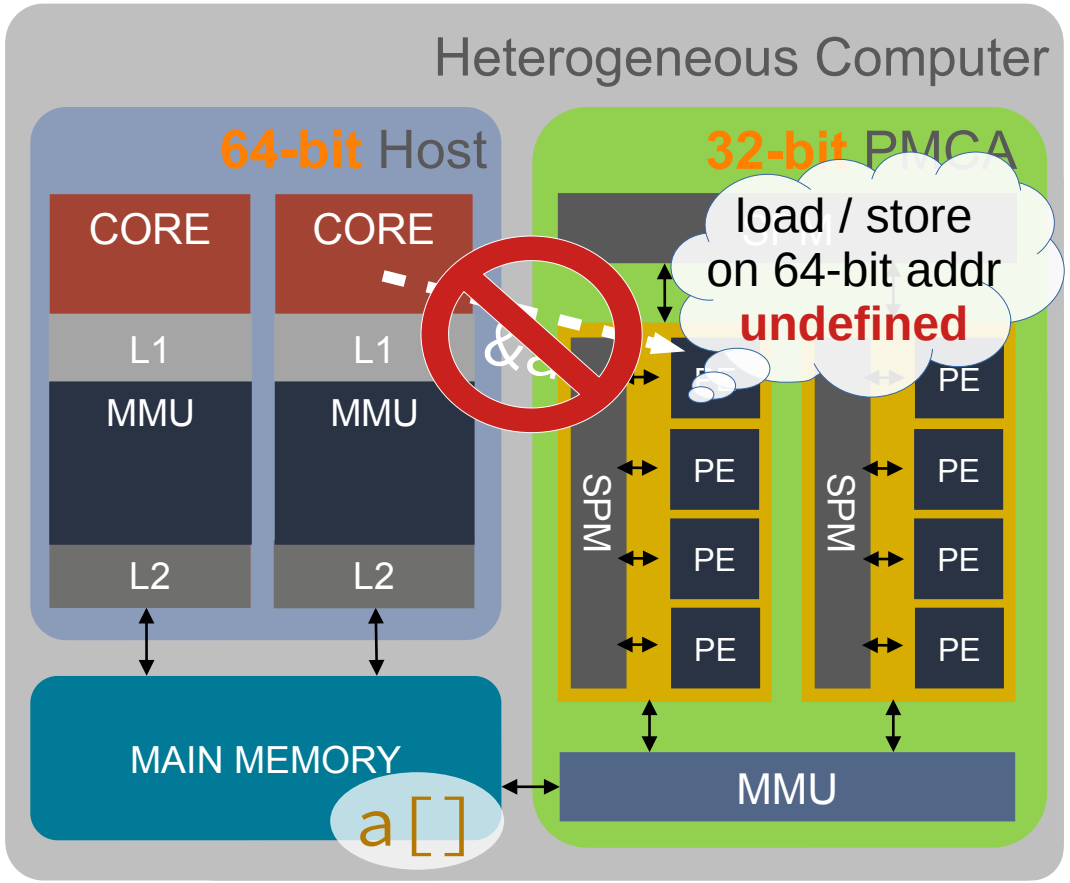
```
void host_function(int n, int* a) {
  #pragma omp target map(tofrom: a[0:n], n)
  {
    accelerator_function(n, a);
  }
}
```



Limitation of State of the Art: Same Data Model for Host and PMCA

- **Limitation** of the **state of the art: 32-bit PMCAs cannot use 64-bit pointers** (neither **hardware** nor **compilers**)!
- **Workaround** variants: **restrict shared address** space to **32 bit** by
  - running entire applications in 32-bit mode [32]
  - restricting shared virt. addresses to 32 bit [25]
  - letting host relay data to 32-bit addressable memory [9,17]
- Workarounds **cannot** implement full ***Shared Virtual Memory*** [29,33,44,58]; thus
  - prohibit sharing of pointer-based data
  - incur unnecessary data movement

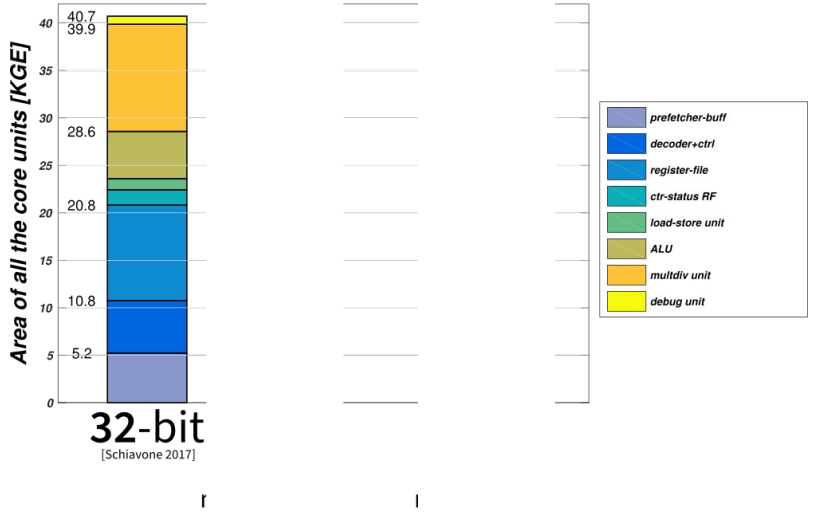
```
void host_function(int n, int* a) {
  #pragma omp target map(tofrom: a[0:n], n)
  {
    accelerator_function(n, a);
  }
```



Why Not Just Use 64-bit Accelerator Cores?

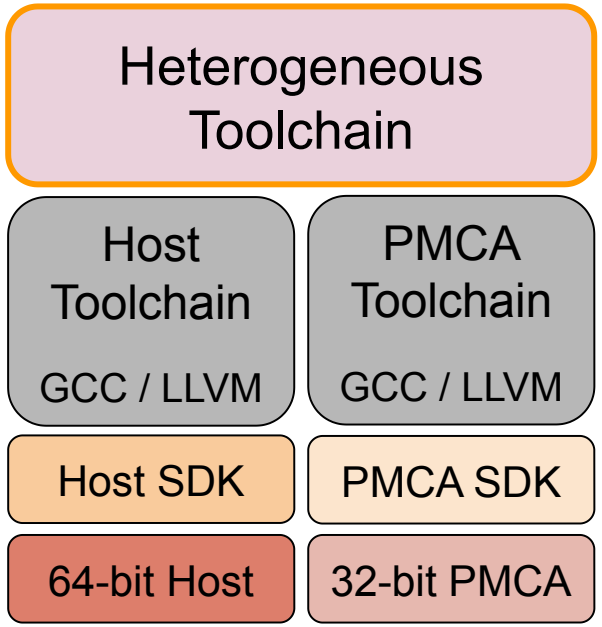
- **Doubling** the **data width** of a core
  - **doubles** the area of the register file, the load-store unit, and the ALU, and
  - **quadratically** increases the area of the **multiplier** and **divider**

➢ Unless your application domain can use the increased data width, you are **wasting**
  - **performance** per area
  - **energy** per computation

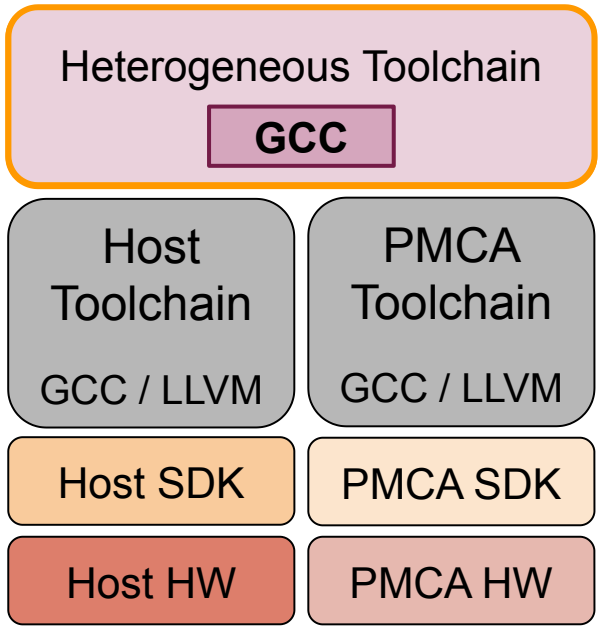


Contributions of Our Work

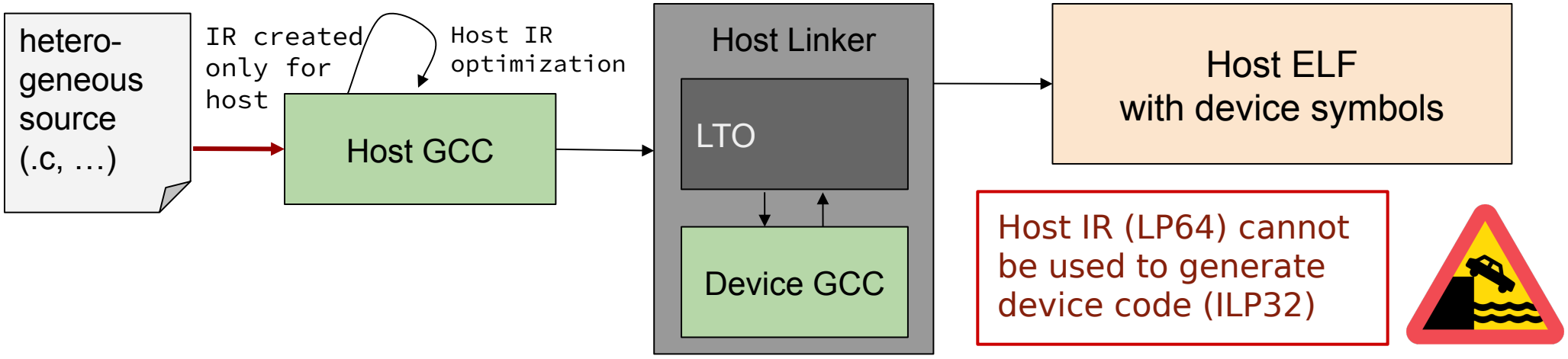
- Our work is the **first to enable transparent off-loading across data models**; meaning
  - **native data types** and **addresses** for Host *and* PMCA
  - **PMCA** can **access** 64-bit **Host address space**
  - **minimal run-time overhead**
  - **transparent to application programmer**
- Discuss **challenges** and **options** on **GCC** and **LLVM**.
- Discuss and evaluate **novel hardware options** for **address extension**.
- For tangibility, we focus on
  - 64-bit Host, 32-bit PMCA
  - OpenMP [44] as widely-used offloading API



Heterogeneous Compilation in GCC

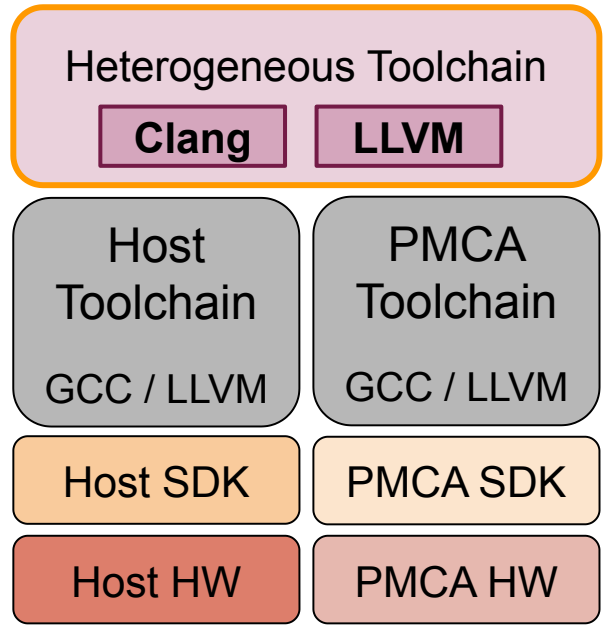


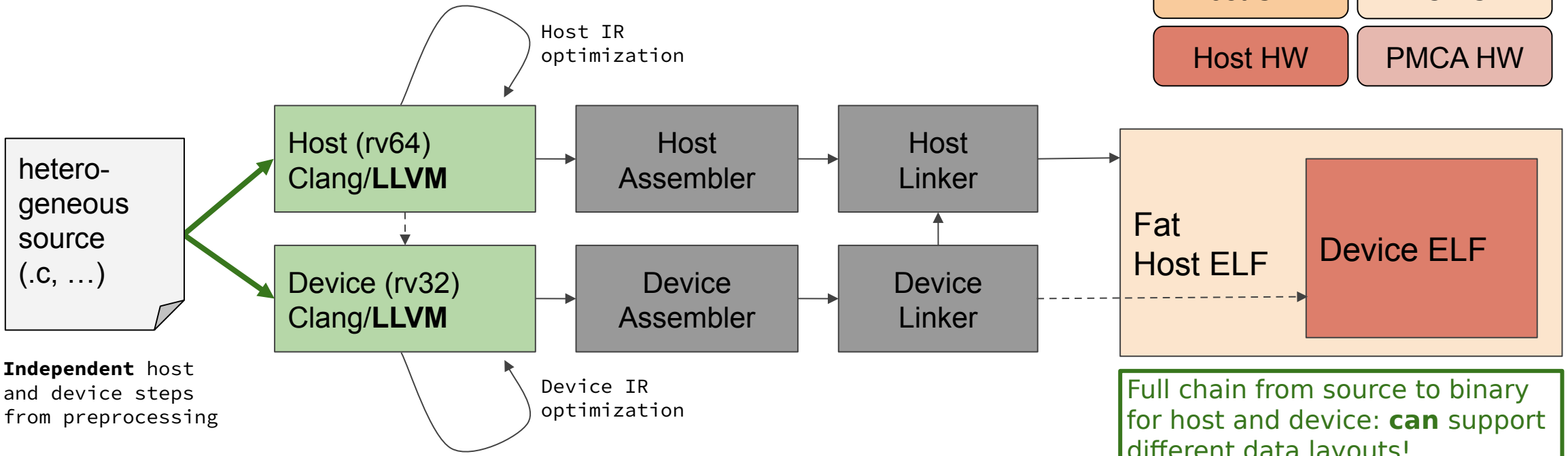
- **Data model directly embedded into GCC's IR**
- **All IR** in **GCC's heterogeneous compilation** uses **data model** of the **Host**
- **GCC cannot handle different data models!**



Heterogeneous Compilation in LLVM

- LLVM to the rescue (?)
- Successful compilation across data models in **LLVM**

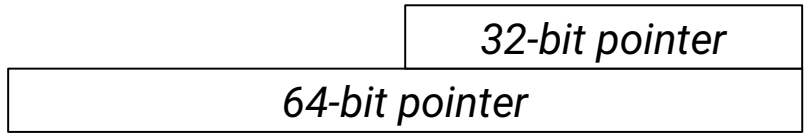




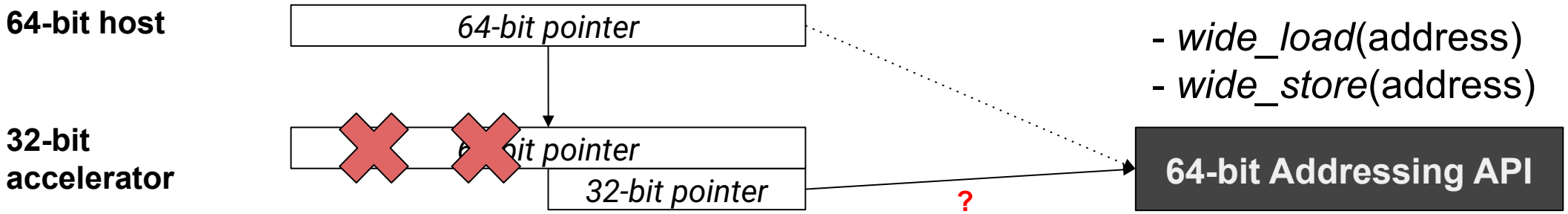
Full chain from source to binary for host and device: **can** support different data layouts!

Sharing 64-bit Pointers with 32-bit Accelerator

- **Mismatch** in data model (**LP64 → ILP32**) must be bridged by the compiler
  - **Fixed width** for **fundamental data types** (e.g., long)
  - **Pointers** remain of **different** size



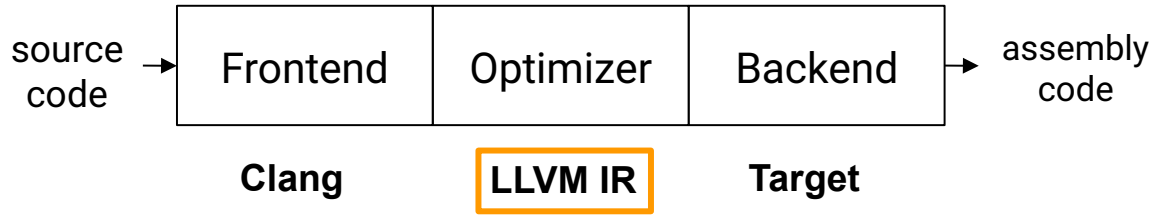
- Assume API to load and store 64-bit addresses on 32-bit accelerator.



- **Challenge**: How to pass 64-bit addresses while native pointers are 32 bit?

Alternatives for Sharing 64-bit Pointers in LLVM

- Application: **fixed-width** integers
  - **Non-intuitive** for the user, solution **not portable**
  - Does **not** work for **arrays** in OpenMP
- Front-end (Clang): **automatic** replacement
  - Tricky to implement
  - **Not transparent** to LLVM IR
- LLVM Intermediate Representation (IR)
  - Separate types for pointers and integers
  - Pointers have **no fixed size** in IR



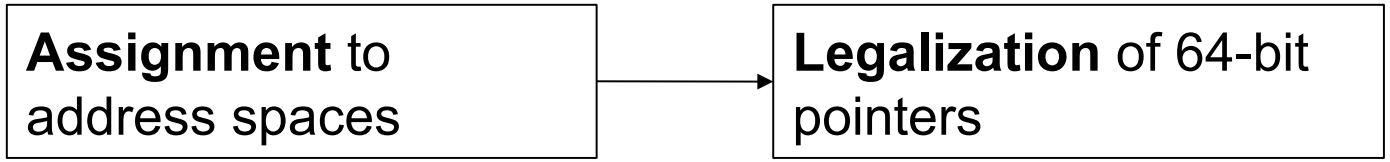

```
void host_function(int* a) {
  uint64_t a_copy = (uint64_t) a;
  #pragma omp target map(tofrom: a_copy)
  {
    int val = wide_load(a_copy);
  }
}
```

```
define i32 @func(i32 %a) {
entry:
  %a.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  %a.load = load i32* %a.addr, align 4
  %add = add i32 %a.load, i32 42
  ret i32 %add
}
```

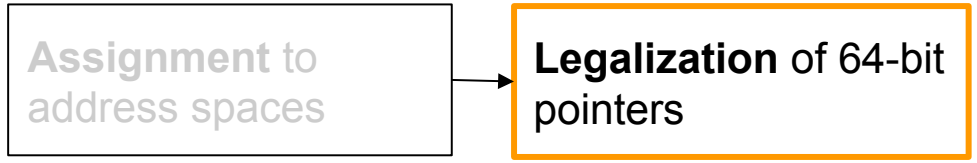
Address Spaces in LLVM

target datalayout = "e-m:e-**p:32:32-p1:64:64**-i64:64-n32-S128"

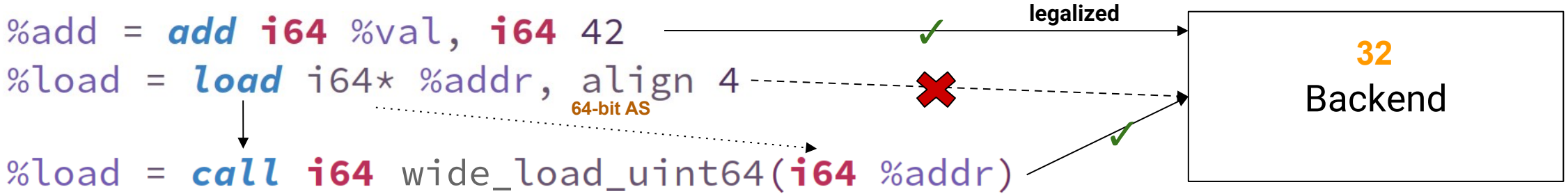
- Pointer sizes set by **address space** via **datalayout string**
- LLVM supports **multiple** address spaces (AS)
  - Generic support of memory regions
  - Used by Nvidia and AMD to represent GPU memory areas
  - Address spaces can be of different size!
- Allows introduction of **64-bit AS** next to native 32-bit AS
- Remaining problems:



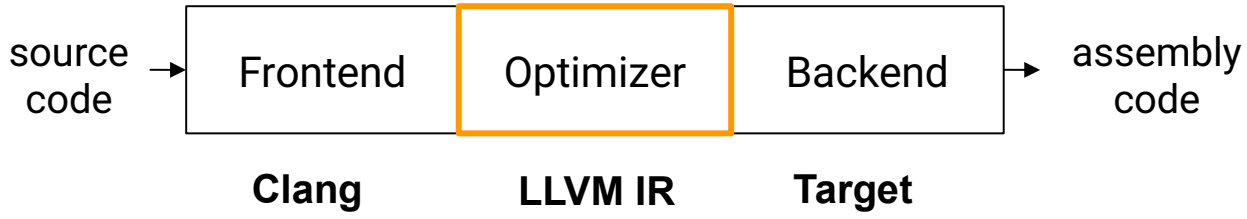
Legalization of 64-bit Pointers



- Backend: no difference between pointers and integers anymore

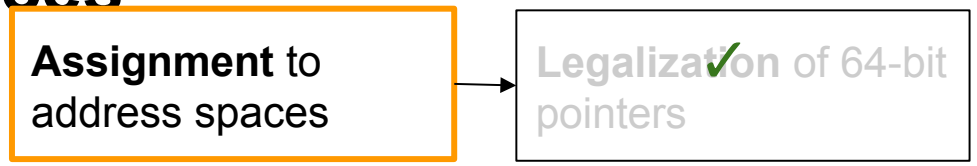


- Insert 64-bit Addressing API calls in optimizer
  - More flexibility towards optimizations
  - Easier implementation as LLVM pass
  - Target-independent approach

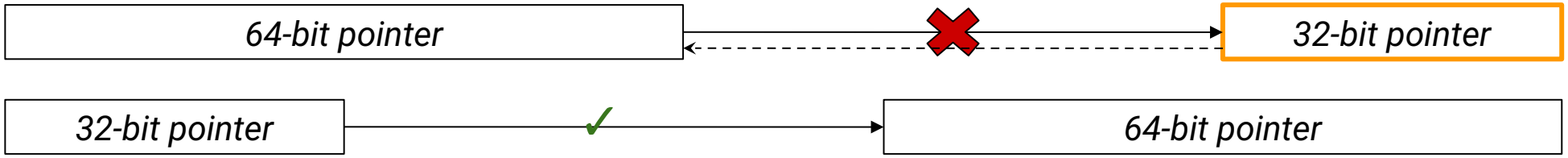


- **native 64-bit arithmetic** for 64-bit pointers in **backend automatically**
- **64-bit load / stores replaced** with 64-bit Addressing API in **optimizer**

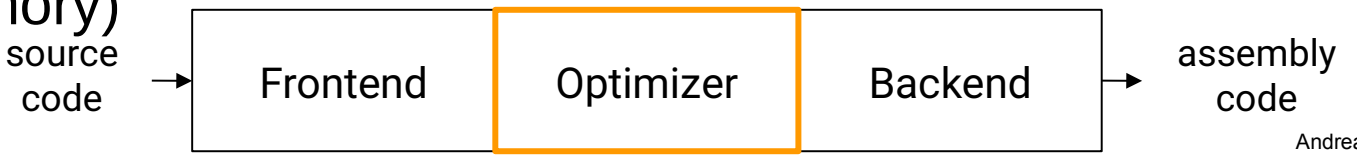
Assignment of Pointers to Address Spaces



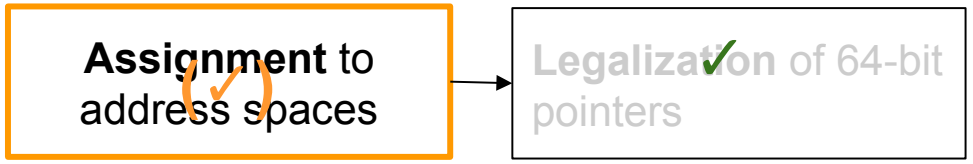
- Pointers by **default** in **generic address space**
- **Observation**: 64-bit pointers are more generic



- **How about 32-bit generic AS** and **converting pointers to 64-bit AS as required?**
  - **Approach:** follow use chains from OpenMP offload entry points
  - **Problem: Full coverage required for correctness** and **difficult** to achieve (storage of pointers in memory)



64-bit Address Space as Default



- **Solution**: Use **64-bit** as **default** address space
- Inherent **correctness** but performance impact → room for **optimizations**
- **Manual** 32-bit assignment in C possible

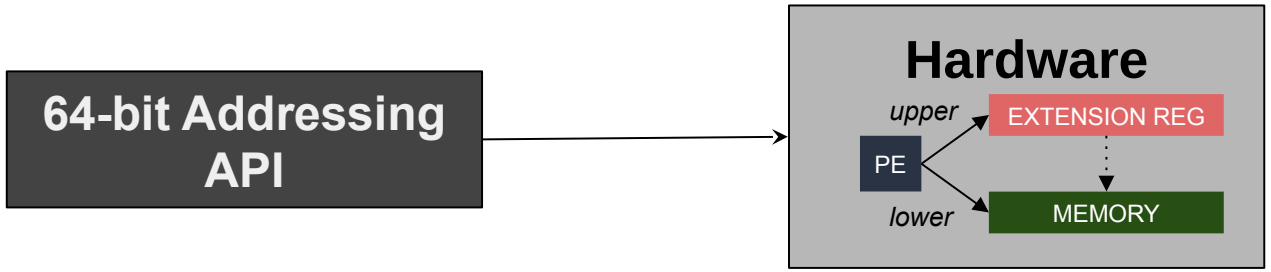
- Functional solution:

```
PMCA: 0x1003f6a8              32-bit pointer
HOST: 0x0000003fffa3bc40      64-bit pointer
```

```c
void host_function(void) {
  int host_array[] = {1, 2};
  #pragma omp target map(tofrom: host_array[0:2])
  {
    __device int* pmca_ptr = (__device int*)pmca_l1malloc(8);
    printf("PMCA: %x\n", (uint32_t)pmca_ptr);
    printf("HOST: %llx\n", (uint64_t)host_array);
  }
}
```

Accelerator Hardware Support

- Different implementation options for 64-bit loads and stores in 32-bit hardware:



1. Wider load and store instructions
2. Additional control and status register (CSR)
3. Memory-mapped external register

Upper-bound on run-time overhead.
Let us evaluate performance!

RISC-V Example

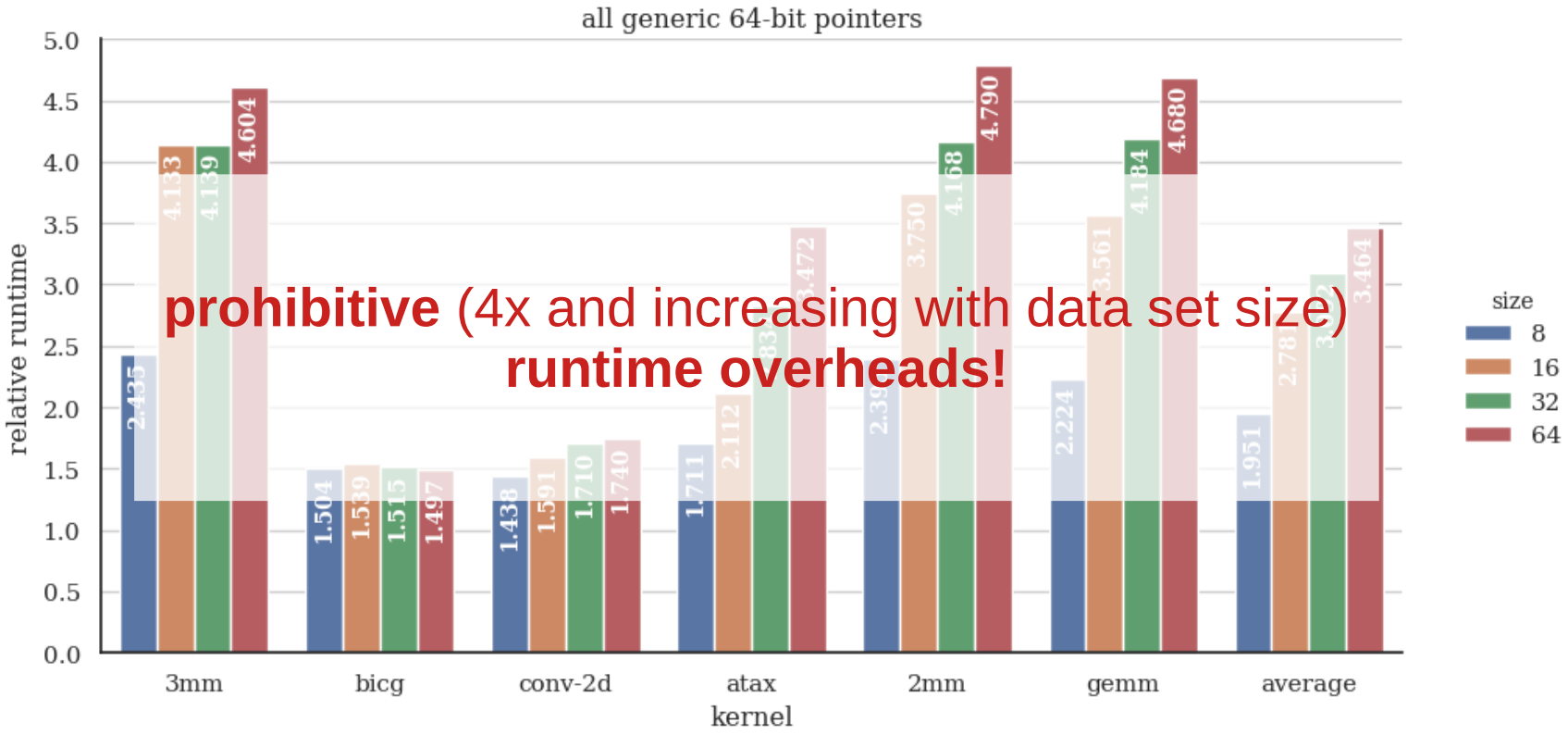
```
csrrci x4, csr_status, 3   // disable interrupts
sw x3, 0(mem_addr_ext)     // set upper half of address
lw x0, 0(x2)               // load lower half of data
csrrw x0, csr_status, x4   // re enable interrupts
```

4 instructions per load/store

Evaluation Setup

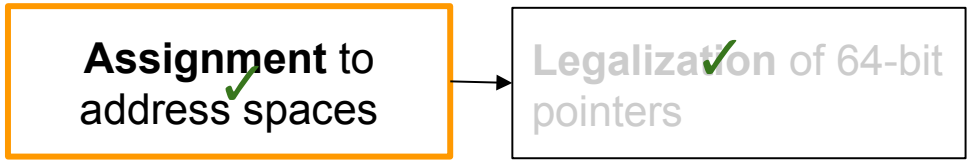
- Cycle-accurate register transfer level (RTL) simulation of PMCA with **8 RISC-V cores** using **PolyBench-ACC** kernels
- Full **optimizations** (-O3) before and after own passes (inlining)
- Data in application memory, manually transferred to **local SPM** by DMA
- **Baseline**: PMCA using its **native 32-bit address space** (as if no Host)

Results without Address Space Assignment Optimization



prohibitive (4x and increasing with data set size)
runtime overheads!

Automatic Optimized Address Space Assignment



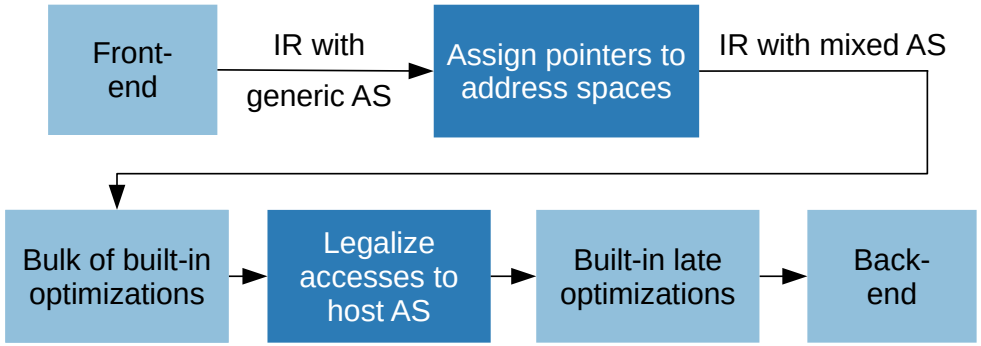
- **Anchor** points followed over **use** chains
- But pointers can also be stored in memory
- **Algorithm 2**: Attempt to **prove** a stored pointer only holds *device* pointers
  - Attempt to follow **uses** of stack-allocated pointers
  - Check if **stores** are only **originating** from device address space
  - Terminate if function exclusively uses pointer **read-only** (OpenMP kernels)
- If yes, invoke **Algorithm 3**: Recursively **update** address space of stored pointer to *device*.
- See paper for formal algorithms and details.

Benchmark Results

Our compiler solution allows 32-bit accelerator to share data in 64-bit address space with negligible overhead!

Conclusion

- Our **compiler solution** allows **accelerator** to share data **outside its native address space** (e.g., 32-bit accelerator and 64-bit AS) **with negligible overhead**.



- **Next steps**:
  - Add support for external functions in standard libraries.
  - Set up hardware prototype on FPGA.
  - Evaluate larger benchmark suites and data sets.