

Cross-Translation Unit Optimization via Annotated Headers

“Header Time Optimization”



William S. Moses

wmoses@mit.edu



Johannes Doerfert

jdoerfert@anl.gov



Writing optimizable code is difficult

- How do we ensure that `norm` is hoisted outside the loop and `normalize` gets vectorized?

```
// Defined inside an external library
double norm(double *A, int n);

void normalize(double *out, double *in, int n) {
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Writing optimizable code is difficult

- Let's add restrict (if in and out alias, calls to norm would be different)

```
// Defined inside an external library
double norm(double *A, int n);

void normalize(double *restrict out, double *restrict in, int n) {
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Writing optimizable code is difficult

- Let's make sure the compiler knows that the input is constant

```
// Defined inside an external library
double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in,
               int n) {
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Writing optimizable code is difficult

- Let's *really* make sure the compiler knows norm is pure

```
__attribute__((pure))
double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in,
               int n) {
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Writing optimizable code is difficult

- Maybe marking it as vectorizable will work?

```
__attribute__((pure))
double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in,
              int n) {
    #pragma clang loop vectorize(enable) interleave(enable)
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```

Writing optimizable code is difficult

- If we mark norm as having the LLVM attributes “readonly” and “argmemonly” this finally happens!

```
__attribute__((fn_attr("readonly"), fn_attr("argmemonly")))  
double norm(double *A, int n);  
  
void normalize(double *restrict out, double *restrict in, int n) {  
    for (int i = 0; i < n; ++i)  
        out[i] = in[i] / norm(in, n);  
}
```


^Note this specific syntax doesn't exist in mainline Clang

This is a problem in real programs

- In the DOE RSBench benchmark adding “readnone” to fast_cexp gives a 7% improvement to the entire program (with another 1% for “unwind”)

```
complex double fast_cexp(double complex z);  
  
...  
Z = ...;  
  
for (int i = w.start; i < w.end; ++i) {  
    sigT += data.poles[nuc][i] * fast_cexp(Z);  
    ...  
}
```


How do we automatically make code optimizable?

Idea! 

LLVM automatically derives these attributes as part of the compilation process, then throws it away when it's done

Let's ensure this information is accessible across translation units

Why not always use LTO?

- Running LTO (even ThinLTO [1]) is a burden on compile times
- LTO may not be available in your build / operating system
- It's often impossible to run LTO on your entire program (e.g. using an external library)

Also, it's interesting to see how much of LTO's speedups come from “easily fixable” mechanisms and provide user's the agency to fix them in source code (making the speedups available to everyone independent from compiler/linker used)

[1] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 111–121. IEEE, 2017

Introducing “Header Time Optimization”

- At the end of the compilation process, denote what derived attributes can be added to functions using LLVM's existing analyses and Attributor [2]
- Modes of operation:
 - Remark Mode: print out optimization remarks specifying attributes that should be added to functions
 - Pipeline Mode: automatically generate a new header file with this new information
 - Diff Mode (in progress): create a diff for original source tree
- **Annotated headers account for half of speedups found by LTO alone!**

Header Time Optimization - Remarks

```
// file1.c  
double fcexp(double *A, int n) { ...
```



clang -Rannotations

```
file1.c:2:1: remark: derived following attributes:
```

```
fn_attr("readonly") arg_attr(0, "nocapture") arg_attr(0,  
"readonly") [-Rannotations]
```

```
double fcexp(double* a, int n) {
```

Header Time Optimization – Pipeline

```
// fileN.c
```

```
// file1.c
```

```
double  
fexp(double *A, int n){  
    ...  
}
```

clang -hto_dir=hto

```
// hto/fileN.h
```

```
// hto/file1.h
```

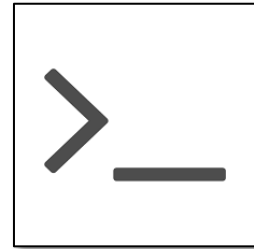
```
attribute((fn_attr("readnone")))  
double fexp(double *A, int n);
```

```
// fileN.c
```

```
// file1.c
```

```
double  
fexp(double *A, int n){  
    ...  
}
```

clang -include=hto/*



executable.o

Header Time Optimization – Pipeline (Library)

```
// libsum.c  
double sum(double *A){  
    ...  
}
```

clang -hto_dir=hto

```
// sum.h  
fn_attr("readnone")  
double sum(double *A);
```



libsum.so

```
// user.c  
double  
fcexp(double *A, int n){  
    sum(A); ...  
}
```

clang user.c -lsum



executable.o

Header files

- HTO creates new files in a given directory that can be included in any C/C++ program (design chosen for easiest experimentation)
- Leverage forward declarations of functions and structs in C/C++

```
struct Vector;  
struct Matrix;  
  
__attribute__((fn_attr("readonly")))  
Vector* matvec(Matrix *M, Vector *B);
```

New attributes for Clang

Not all LLVM attributes are representable with existing Clang attributes

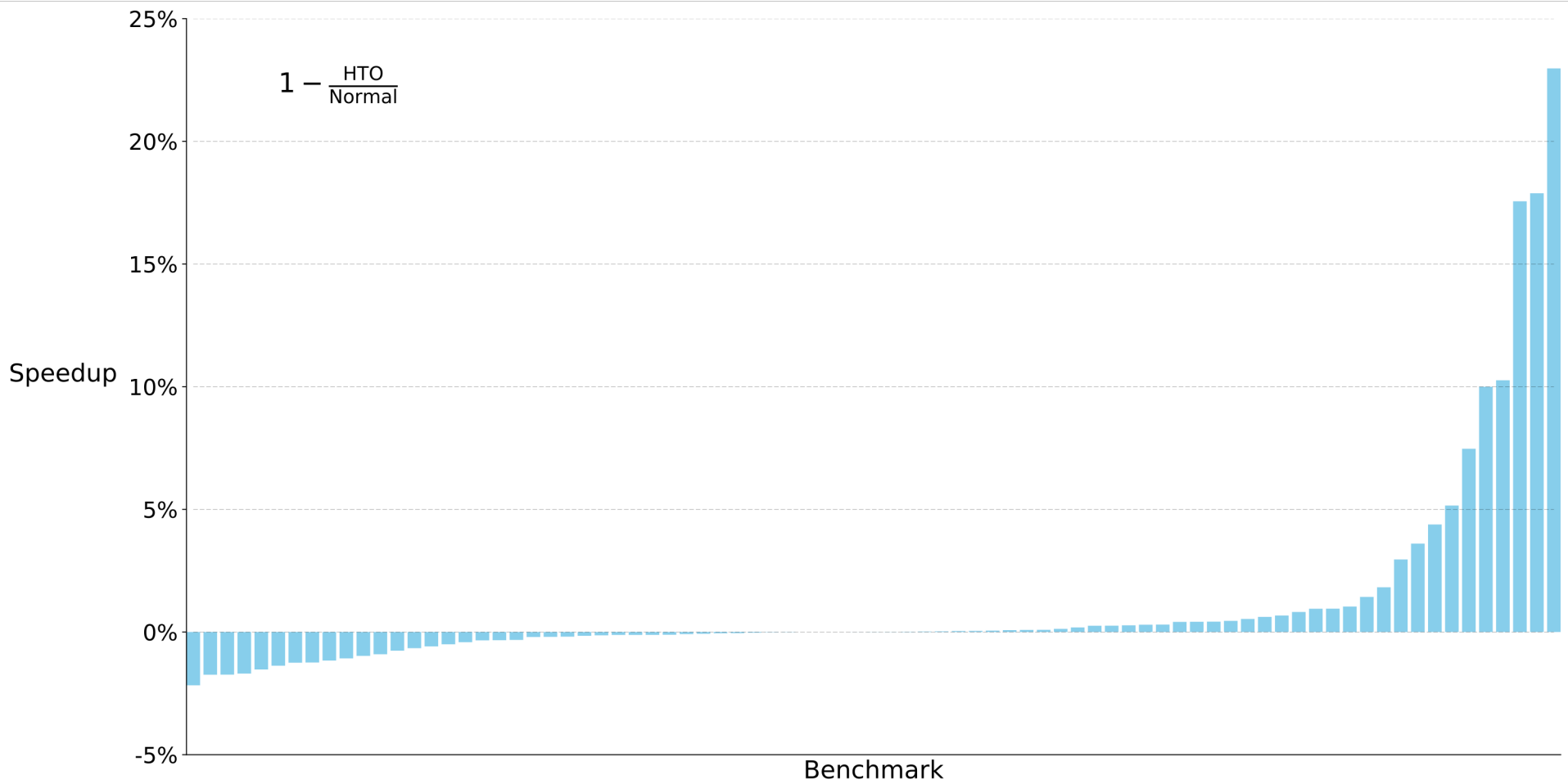
Created a generic way to represent LLVM attributes in Clang

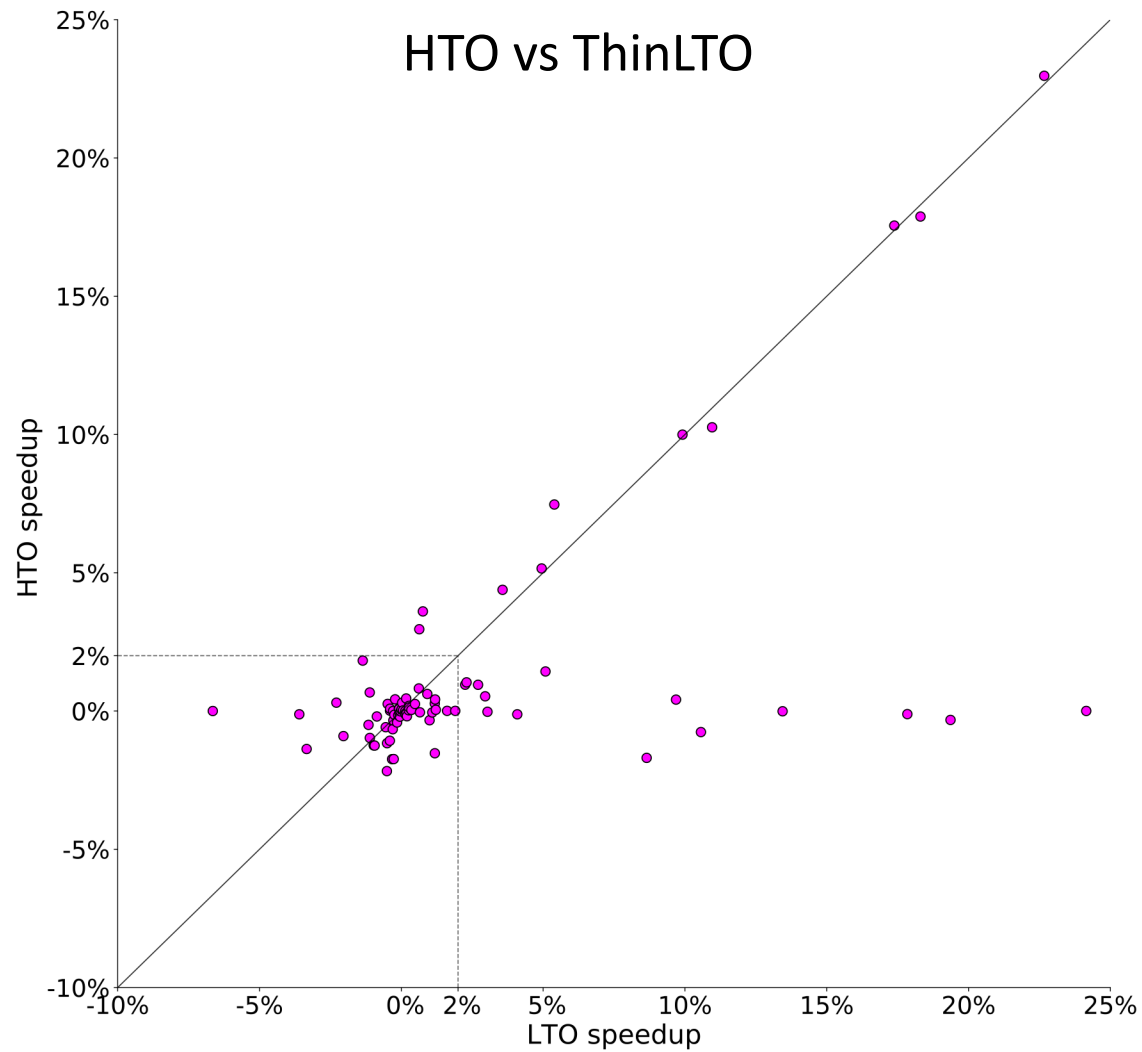
```
__attribute__((fn_attr("readonly")))  
  
__attribute__((arg_attr(1, "readonly")))  
__attribute__((arg_attr(B, "readonly")))  
  
__attribute__((ret_attr("noalias")))  
  
double* vector_add(double *A, double *B, int len);
```


Experiments

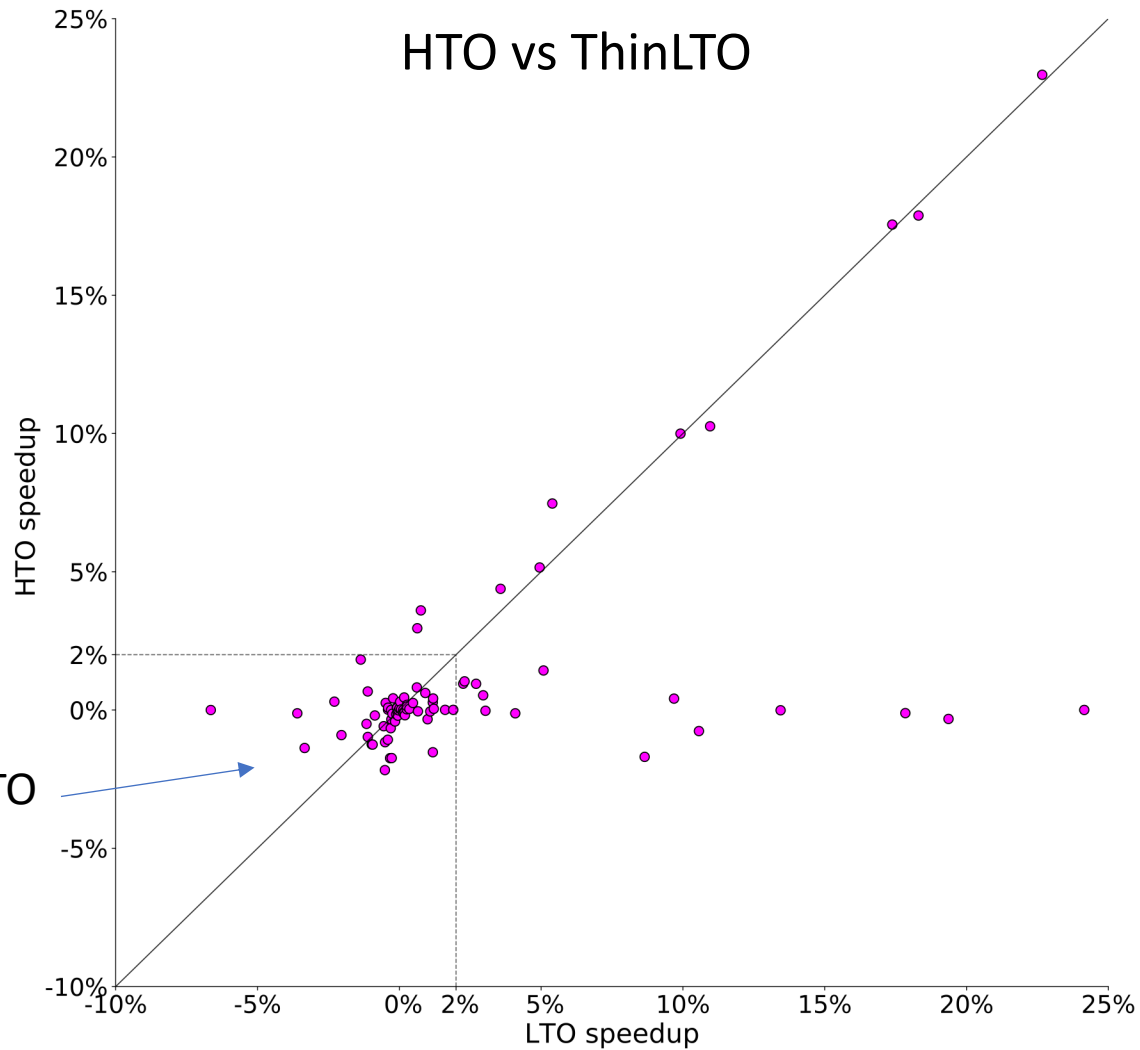
- Multi-source benchmarks in LLVM test suite
 - DOE C/C++ Proxy Apps, Bit Stream benchmark Suite, TSVC test suite, NIST SciMark suite, RNA alignment application, & more
 - Selected tests with runtime > 0.5 seconds
 - 84 tests
- Run min of 10 on quiesced AWS c4.8xlarge
 - Disabled turboboost, hyperthreading
- Compared normal run (with -O3) vs HTO vs ThinLTO

Percentage speedup of HTO over Normal

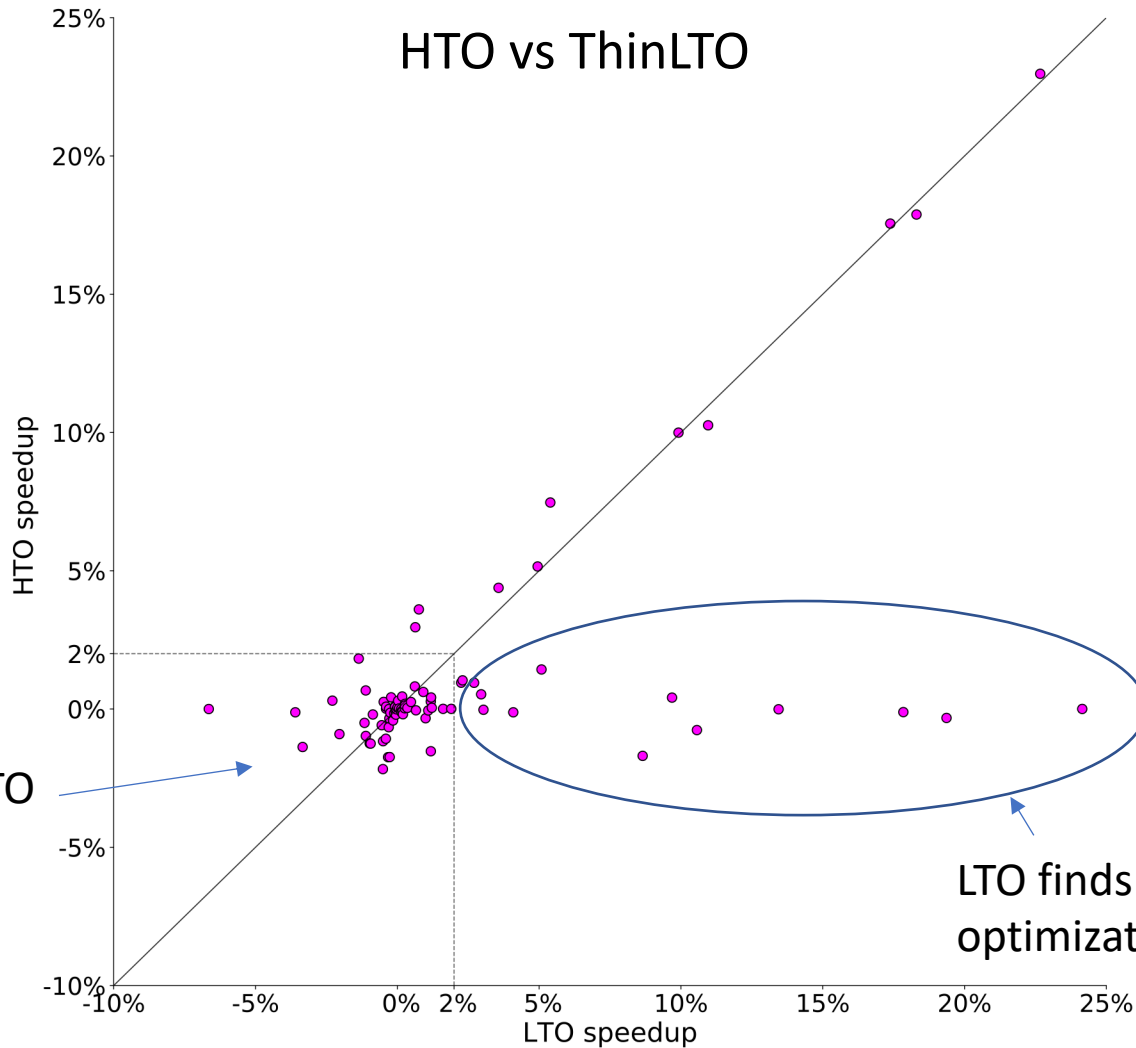




HTO vs ThinLTO



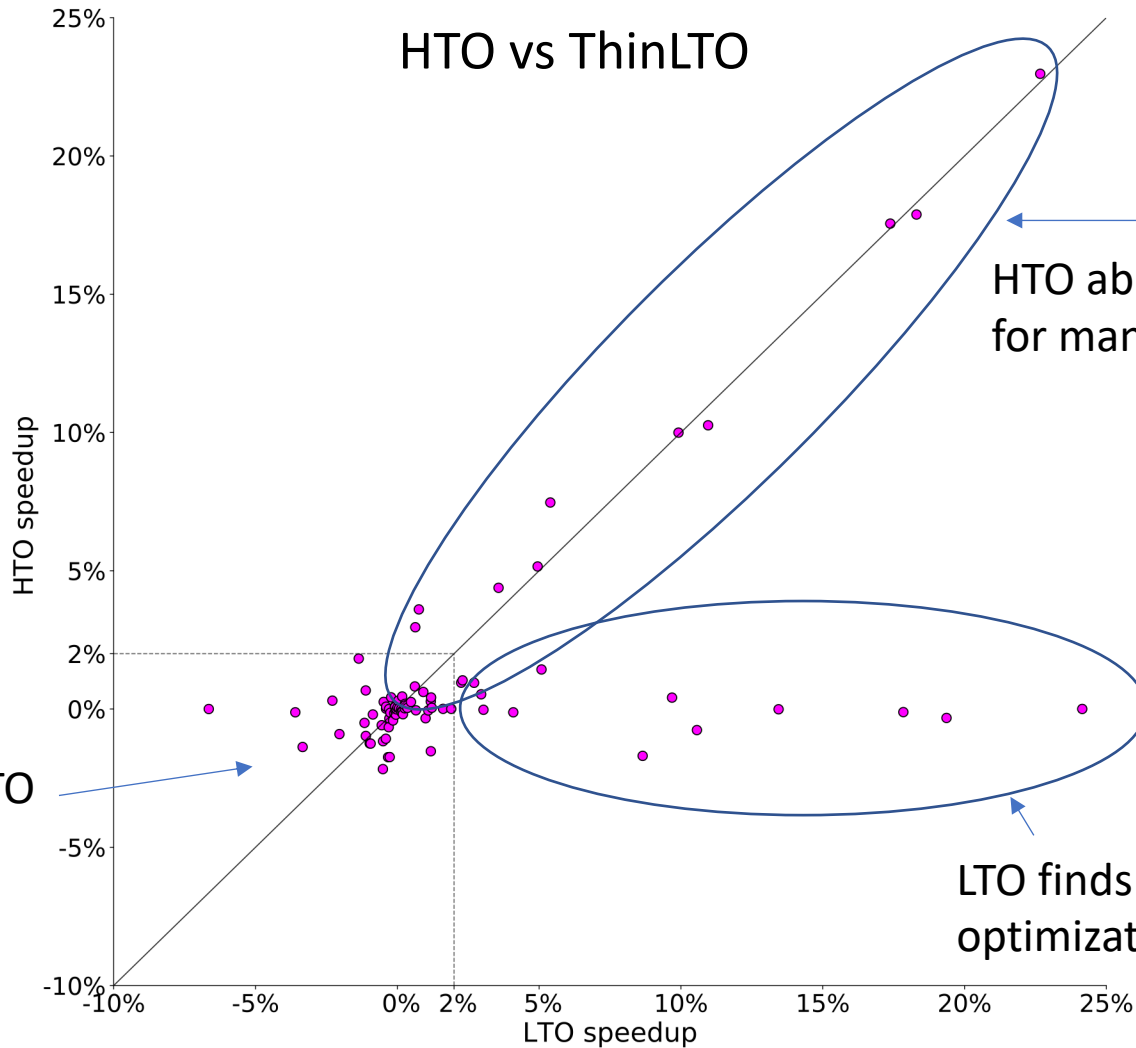
HTO vs ThinLTO



Neither HTO nor LTO
find speedups

LTO finds some
optimizations HTO doesn't

HTO vs ThinLTO

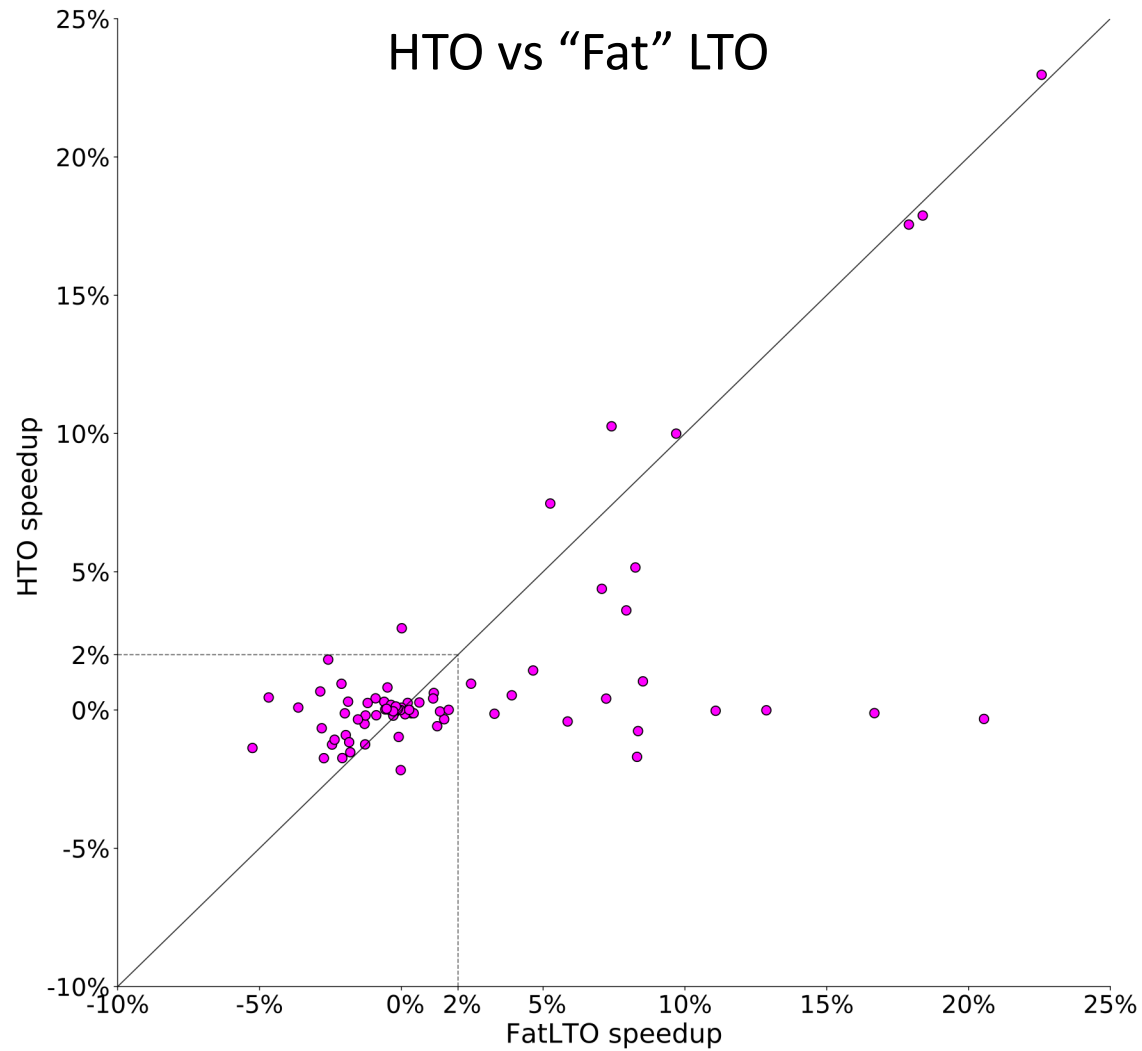


HTO able to match LTO
for many benchmarks

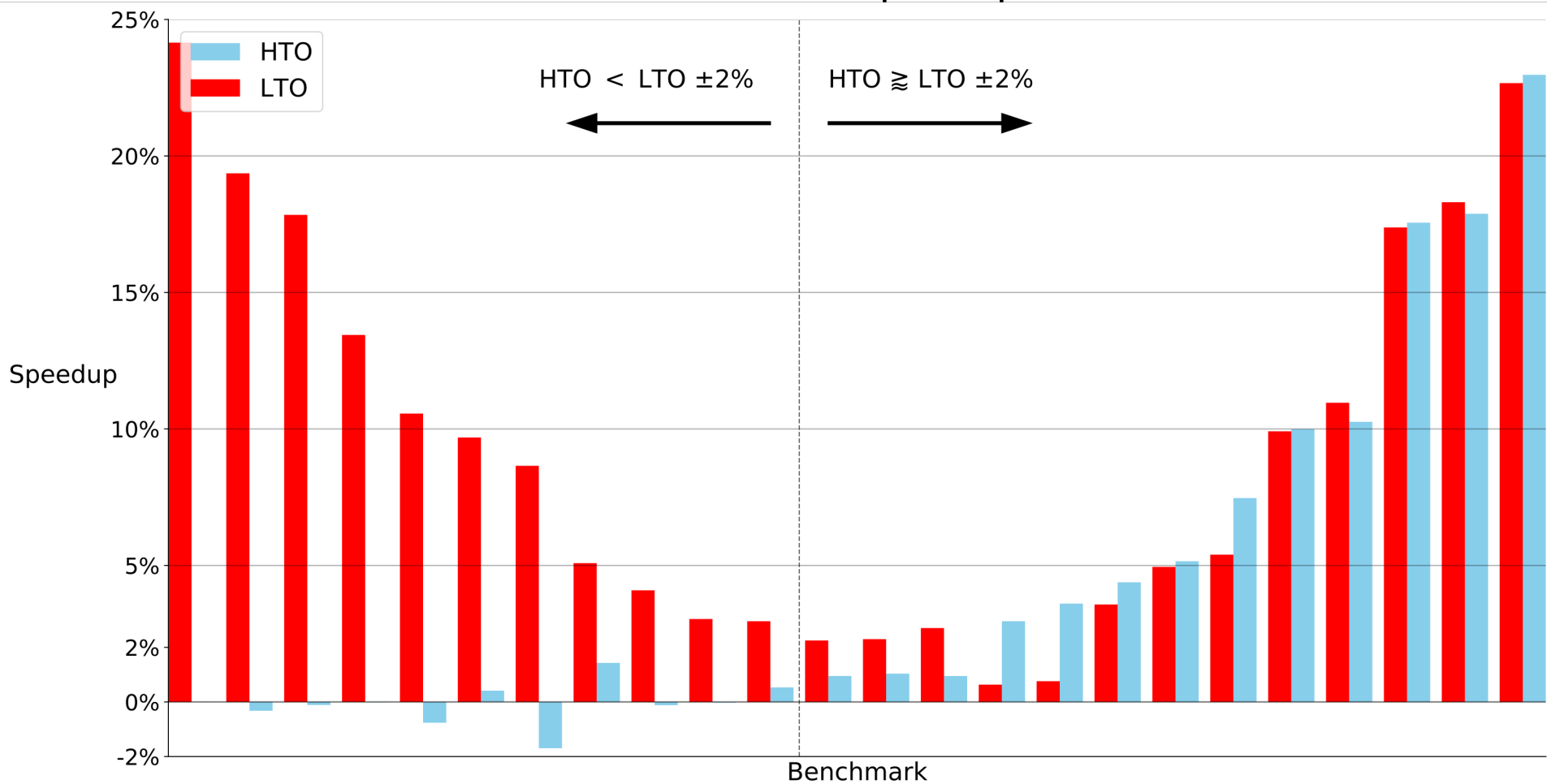
LTO finds some
optimizations HTO doesn't

Neither HTO nor LTO
find speedups

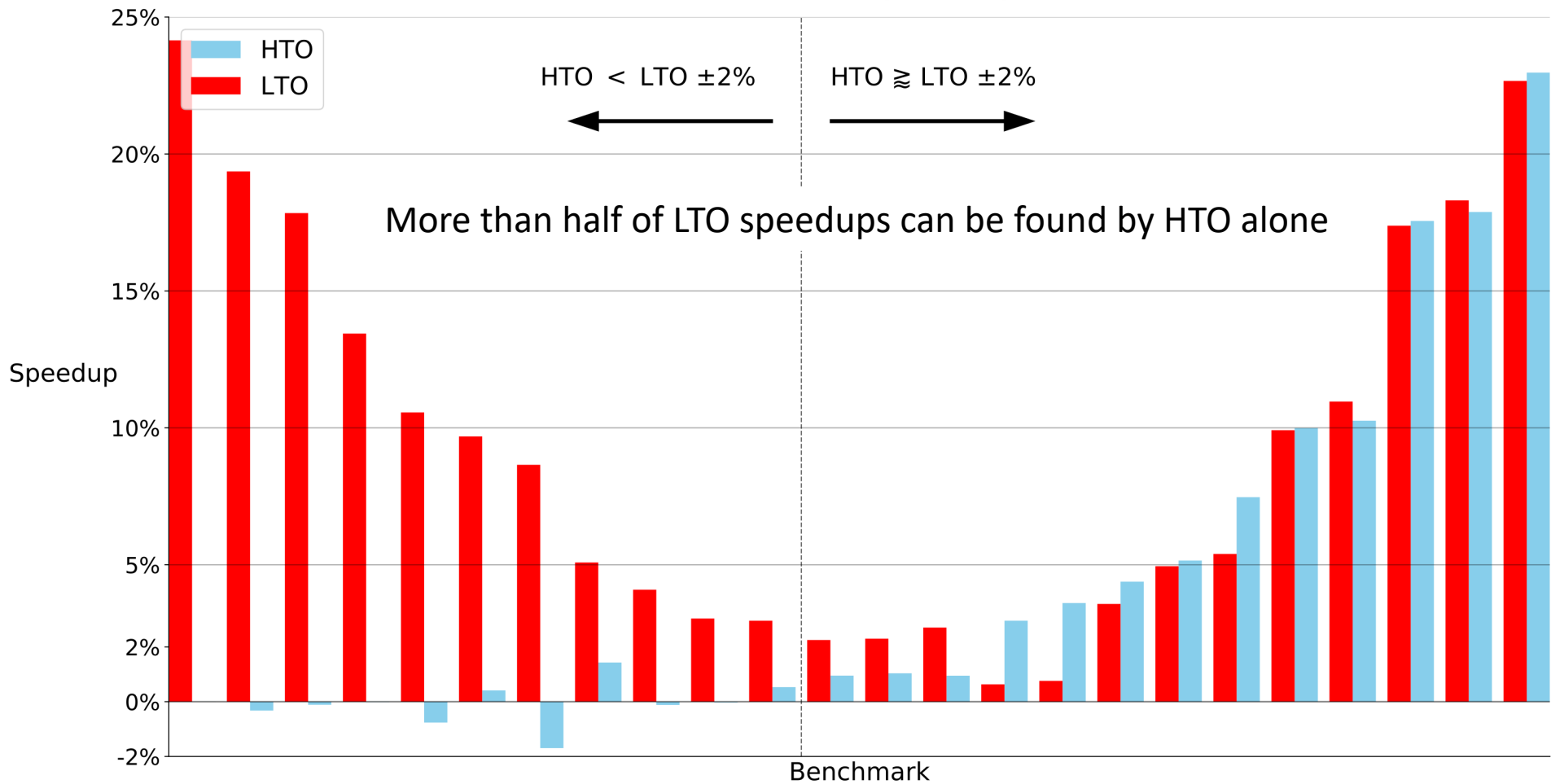
HTO vs “Fat” LTO



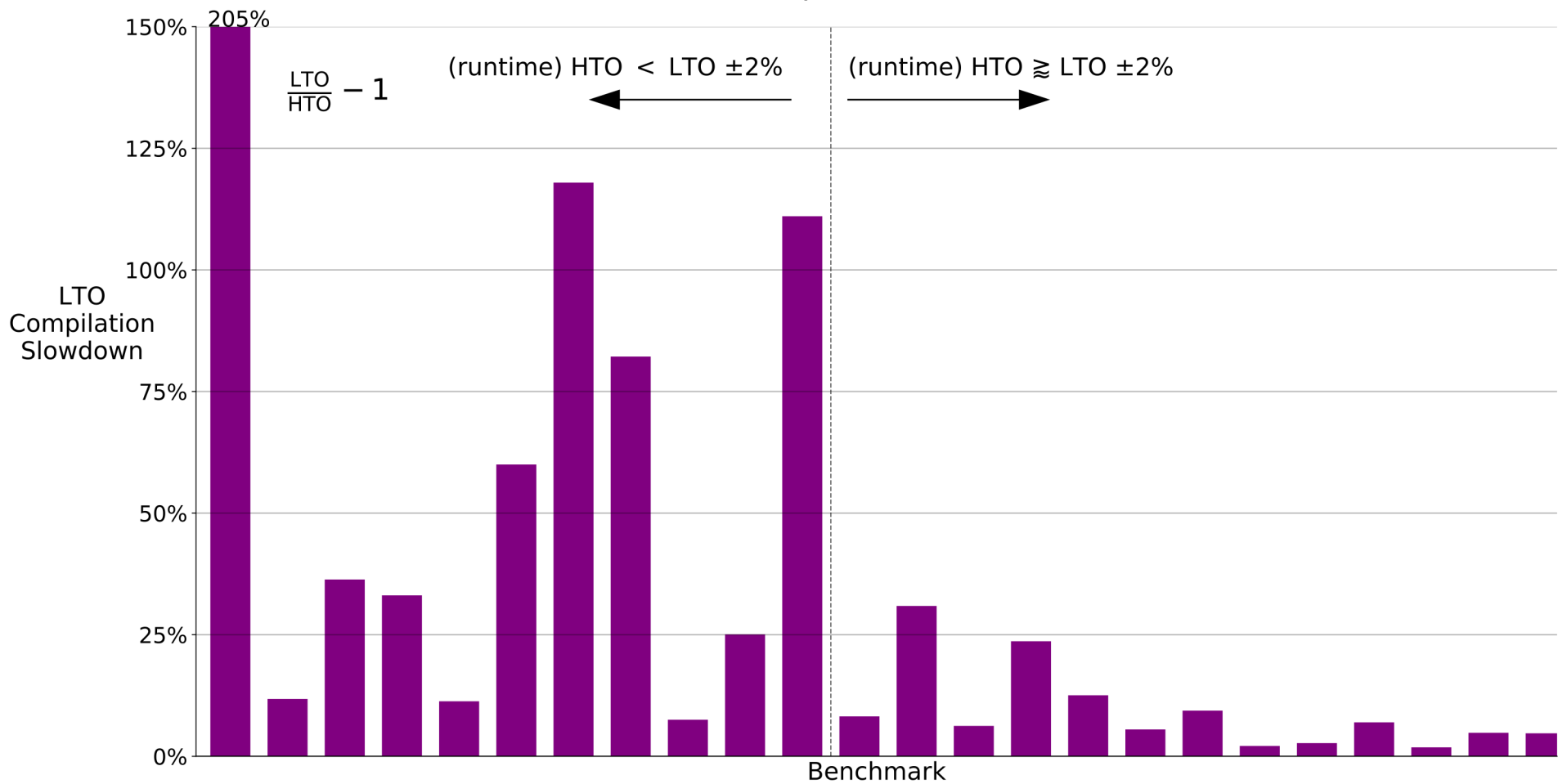
HTO vs ThinLTO Where Speedup Exists



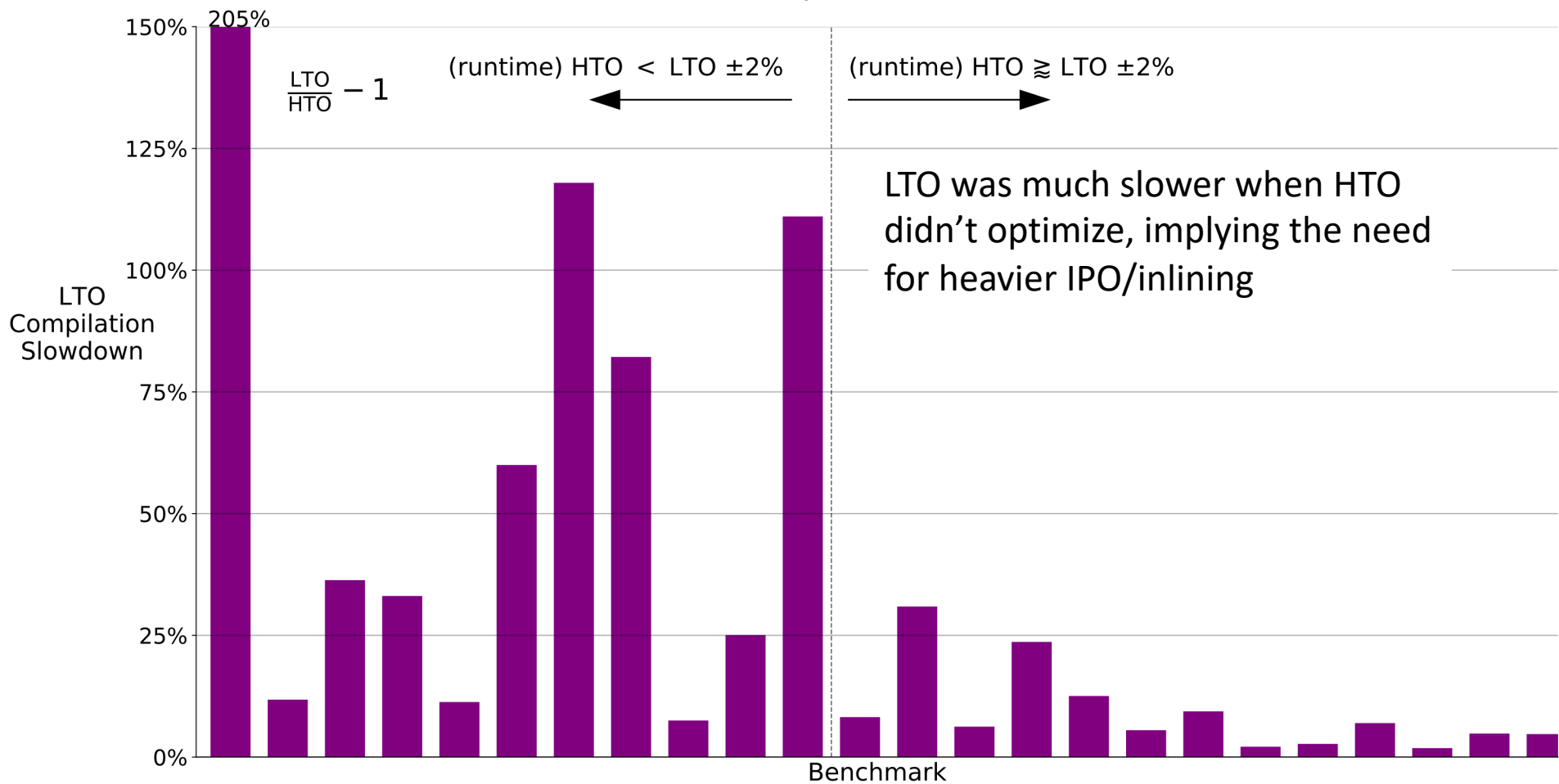
HTO vs ThinLTO Where Speedup Exists



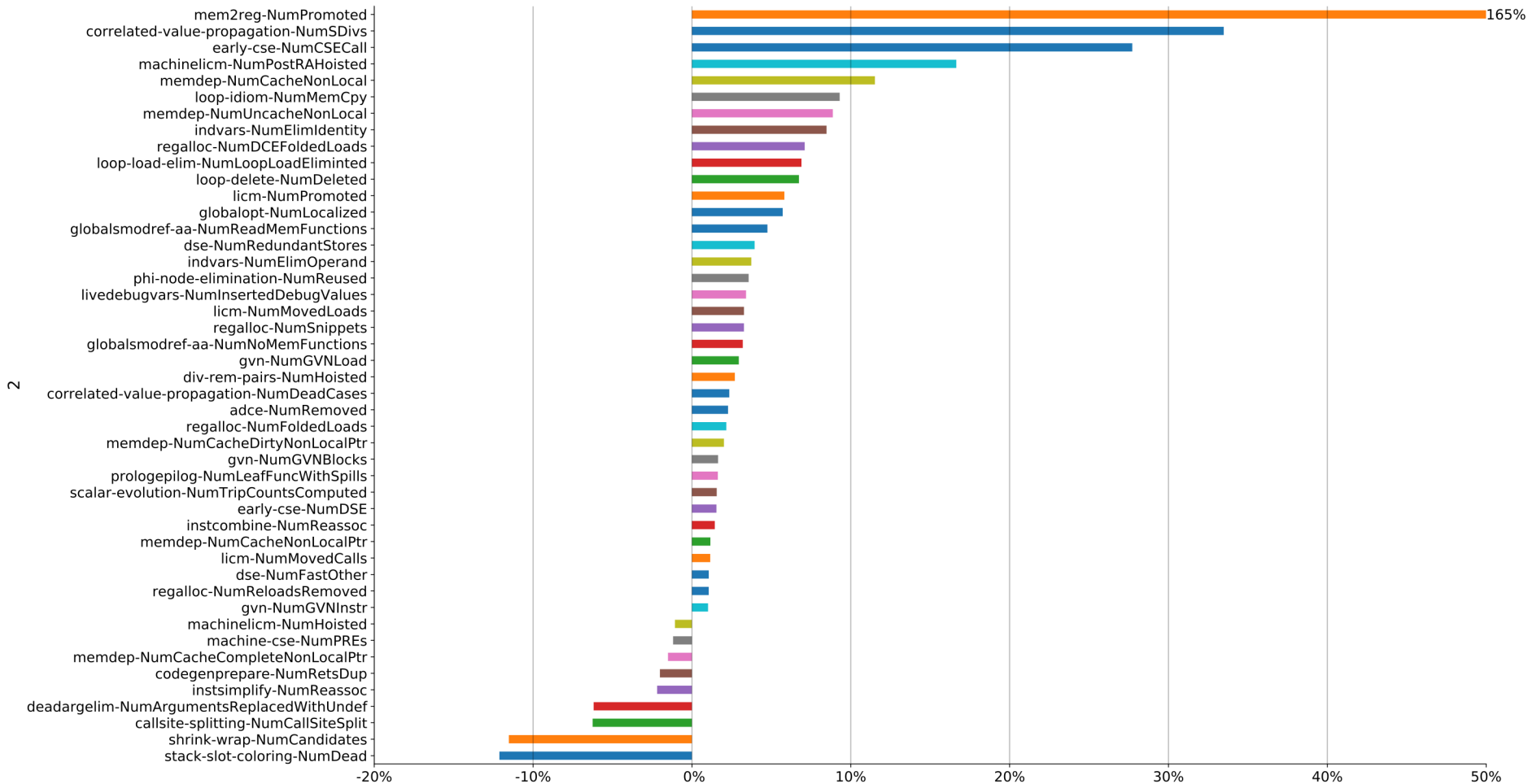
HTO vs ThinLTO Compile and Link Times



HTO vs ThinLTO Compile and Link Times



Optimization Statistics with HTO



Takeaways

- HTO (and LTO) can have meaningful performance gains for programs
 - In practice programs don't have the attributes they should
 - Lack of attributes hinders optimizations
- HTO provides tools to automatically integrate these attributes into your programs / libraries with an additional compiler flag
- Simply propagating attributes (via HTO) accounts for ~50% of the speedups seen by LTO
 - Other 50% likely need IPO/inlining

Present Limitations & Future Work

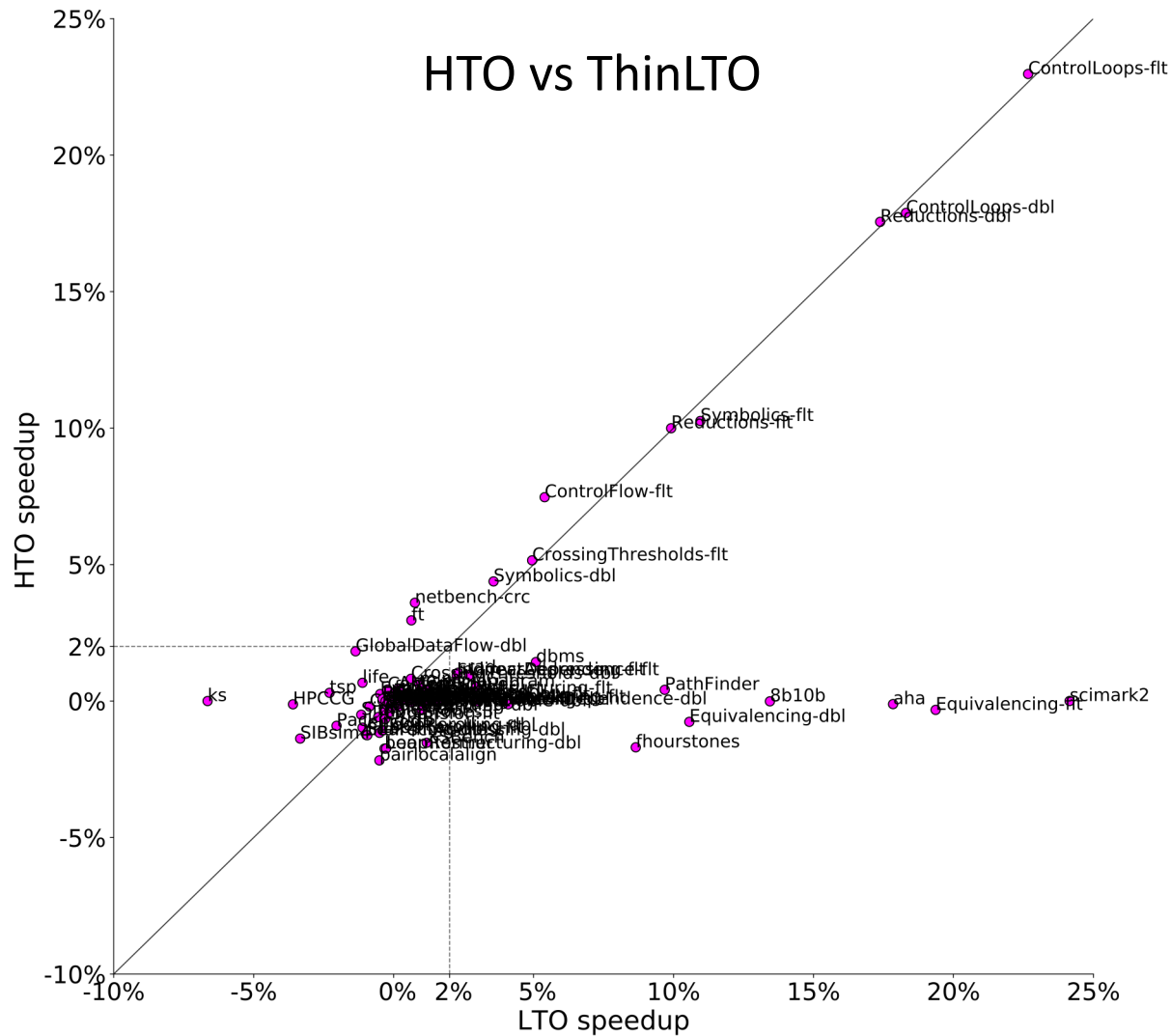
- No anonymous structs (have a script to automatically generate random names)
- No attributes generated for C++ member functions (since can't forward declare)
- No attributes generated for array type (not pointer type) of struct/classes (type `mystruct[3]` is incomplete ahead of time)
- Allow outputting a diff (resolves above and allows for more performance gains)
- Should generate standard C/C++ attributes when they exist

Conclusions

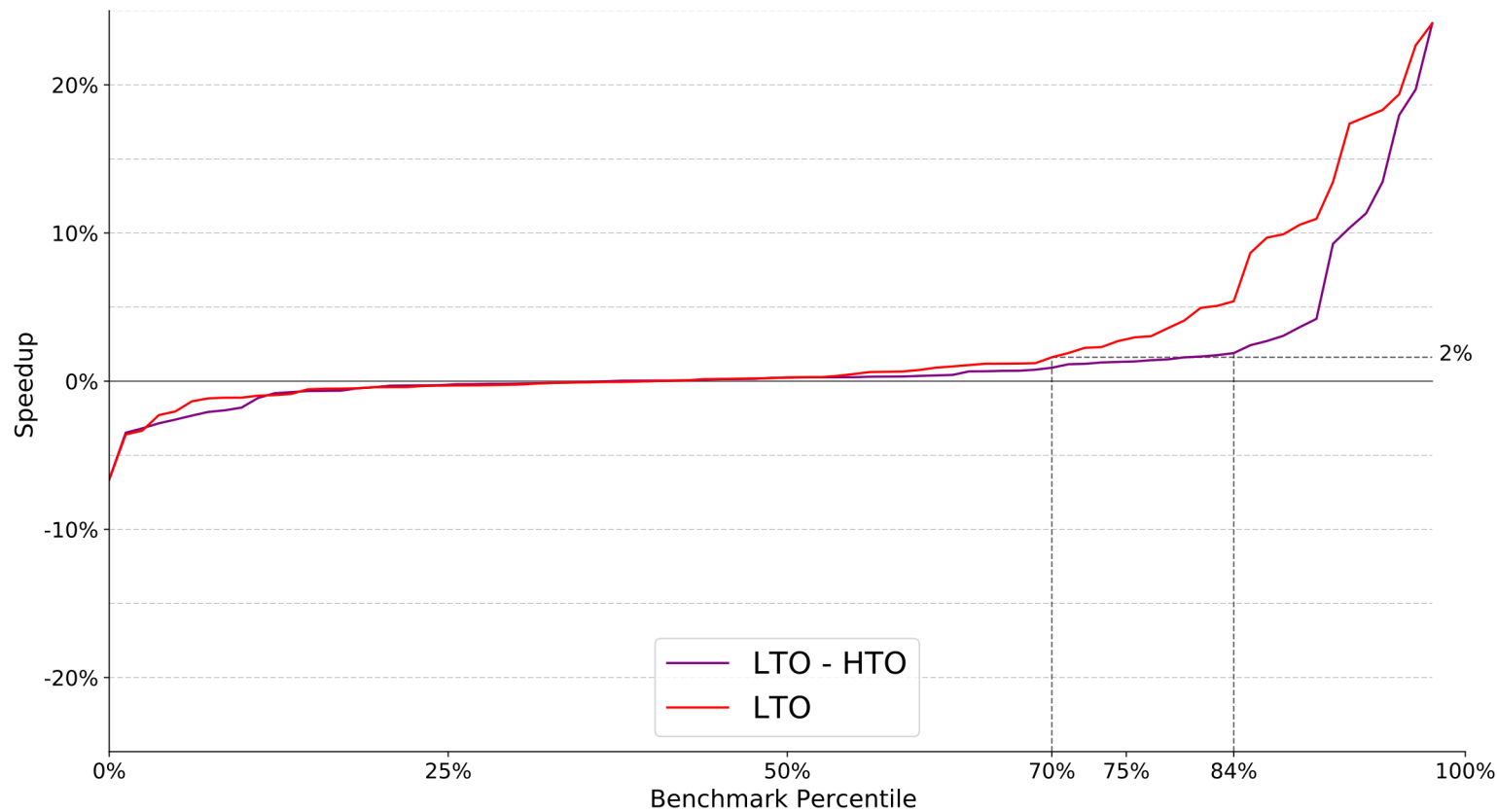
- Properly annotating programs can make real performance difference
- Writing annotations manually can be difficult
- We provide three mechanisms to remedy:
 - Support for writing LLVM annotations in C/C++
 - Optimization remark to point out missing annotations
 - Pipeline to automatically inject missing annotations into programs
- Code is available on Github (github.com/wsmoses/LLVM-HTO), plans to upstream once cleaned up (and submitted for publication)
- Thanks to DOE CSGF/Exascale project and Google Summer of Code for funding the project

Appendix

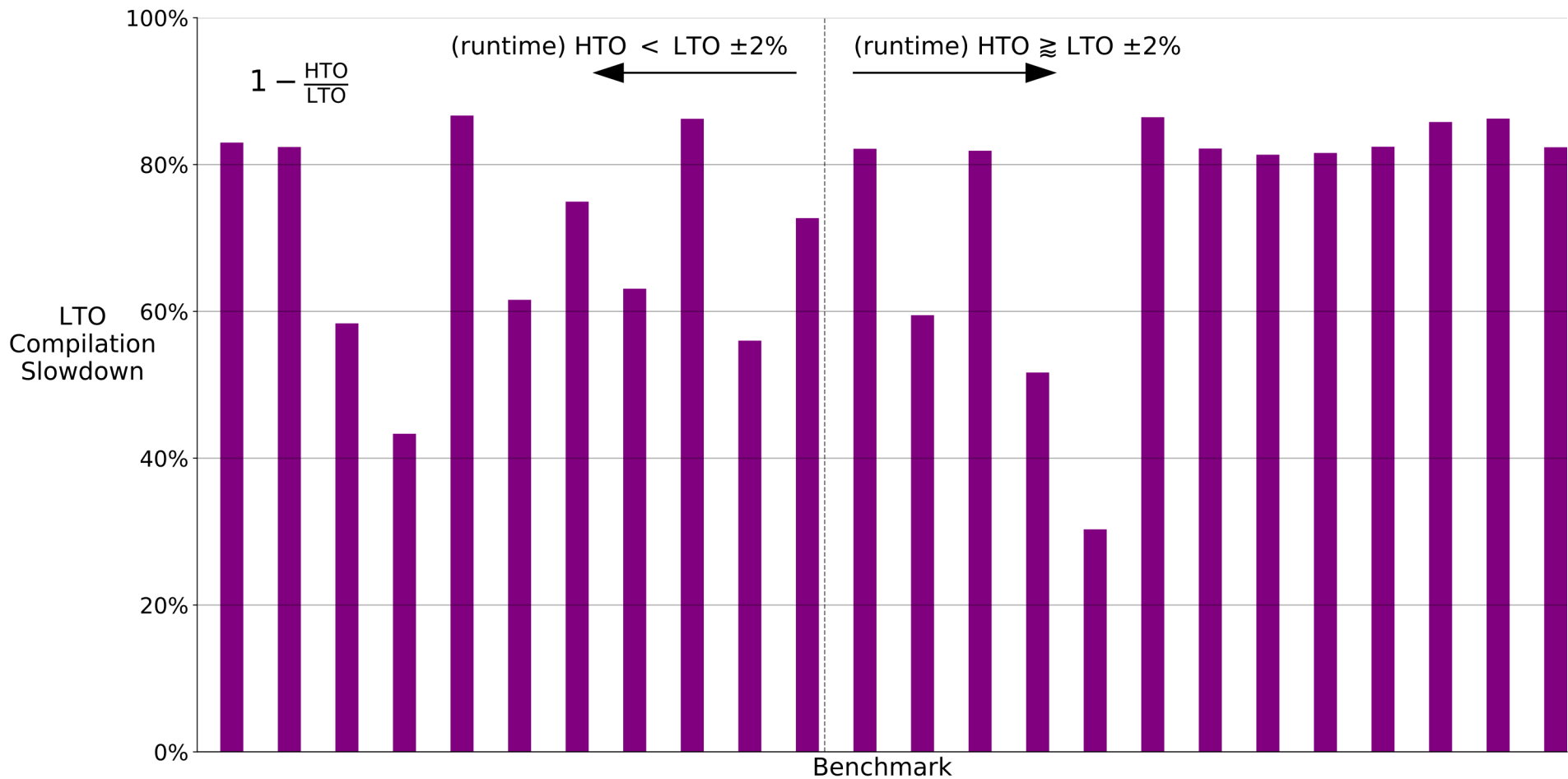
HTO vs ThinLTO



HTO vs ThinLTO



Optimized HTO vs ThinLTO Compile and Link Times (Estimated)



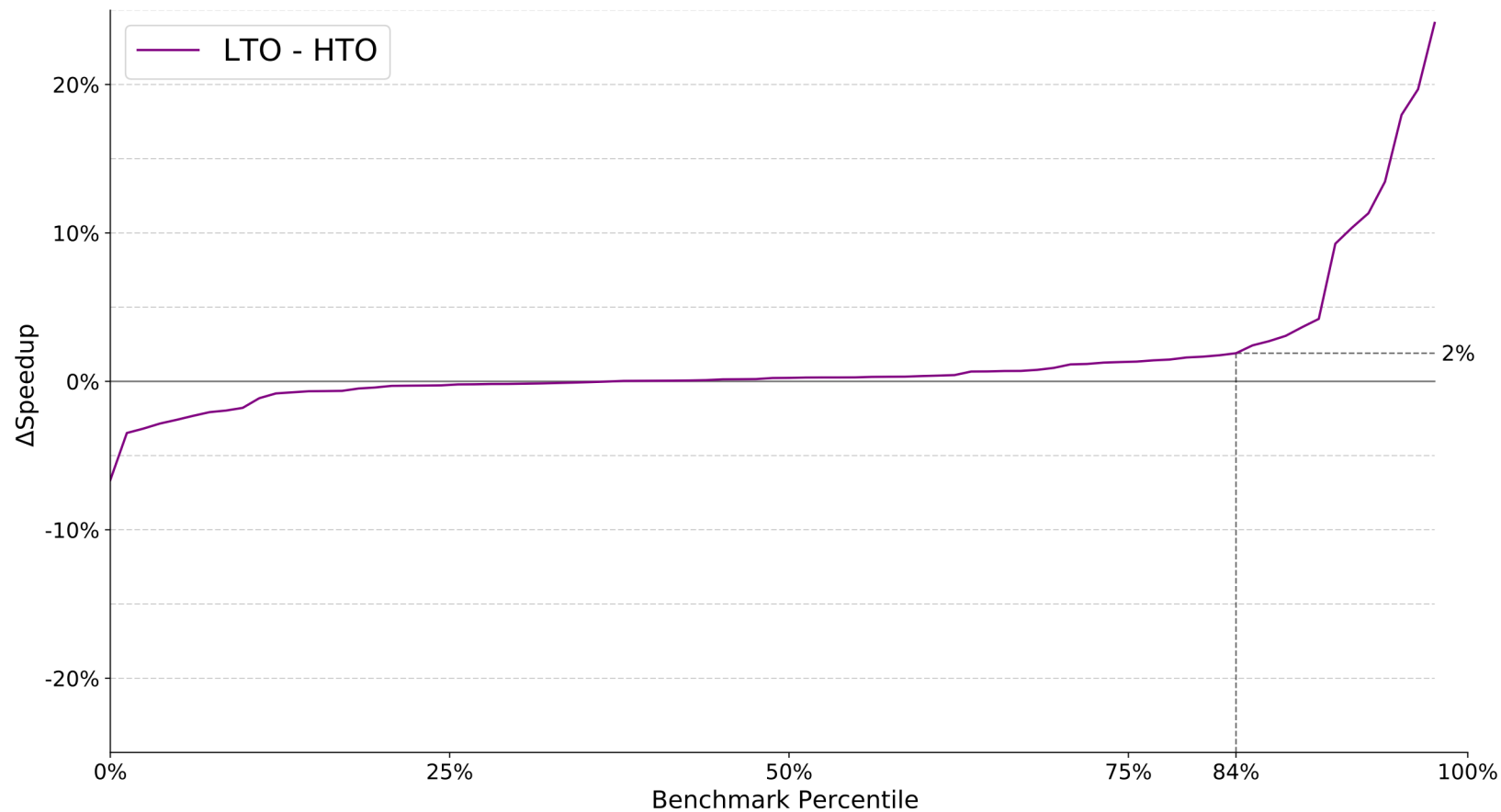
Headers Aren't Optimized Yet

```
1 #ifdef __cplusplus
2 extern "C" {
3 #endif
4
5 __attribute__(( fn_attr("nofree") ,fn_attr("norecurse") ,fn_attr("nosync") ,fn_attr("nounwind") ,
6   n_attr("uwtable") ,fn_attr("writeonly") ,arg_attr(1, "nocapture") ,arg_attr(1, "writeonly")
7 ))
8 int set1d(float arr[32000], float value, int stride);
9
10 #ifdef __cplusplus
11 }
12 #endif
13 #ifdef __cplusplus
14 extern "C" {
15 #endif
16
17 __attribute__(( fn_attr("nofree") ,fn_attr("norecurse") ,fn_attr("nosync") ,fn_attr("nounwind") ,
18   n_attr("uwtable") ,fn_attr("writeonly") ,arg_attr(2, "nocapture") ,arg_attr(2, "writeonly")
19 ))
20 int set1ds(int _n, float arr[32000], float value, int stride);
21
22 #ifdef __cplusplus
23 }
24 #endif
```

Compiler, not Programmer should Optimize Code

- Easier to maintain
- Costly (time or financially) to manually optimize
 - Legacy code base
- Insufficient expertise
- Can't modify source code
 - Calling external library
 - Called by a user program

HTO vs ThinLTO



Writing Optimizable Code is Difficult

- What do we need to do to ensure norm is hoisted outside the loop?

```
__declspec((noalias)) // == argmemonly in llvm, not combinable
__attribute__((pure))
double norm(const double *A, int n);

void normalize(double *restrict out, const double *restrict in,
               int n) {
    #pragma clang loop vectorize(enable) interleave(enable)
    for (int i = 0; i < n; ++i)
        out[i] = in[i] / norm(in, n);
}
```