Integrating LLVM with the PGI Fortran, C and C++ compilers

Third Workshop on the LLVM Compiler Infrastructure in HPC

Doug Miles, PGI Compilers & Tools, NVIDIA Corporation, 14 November, 2016

PGI®

PGI Fortran , C & C++ Compilers

Optimizing, SIMD Vectorizing, OpenMP

Accelerated Computing Features

OpenACC Directives

CUDA Fortran

Multi-Platform Solution

x86-64 and OpenPOWER CPUs,
Tesla and Radeon GPUs

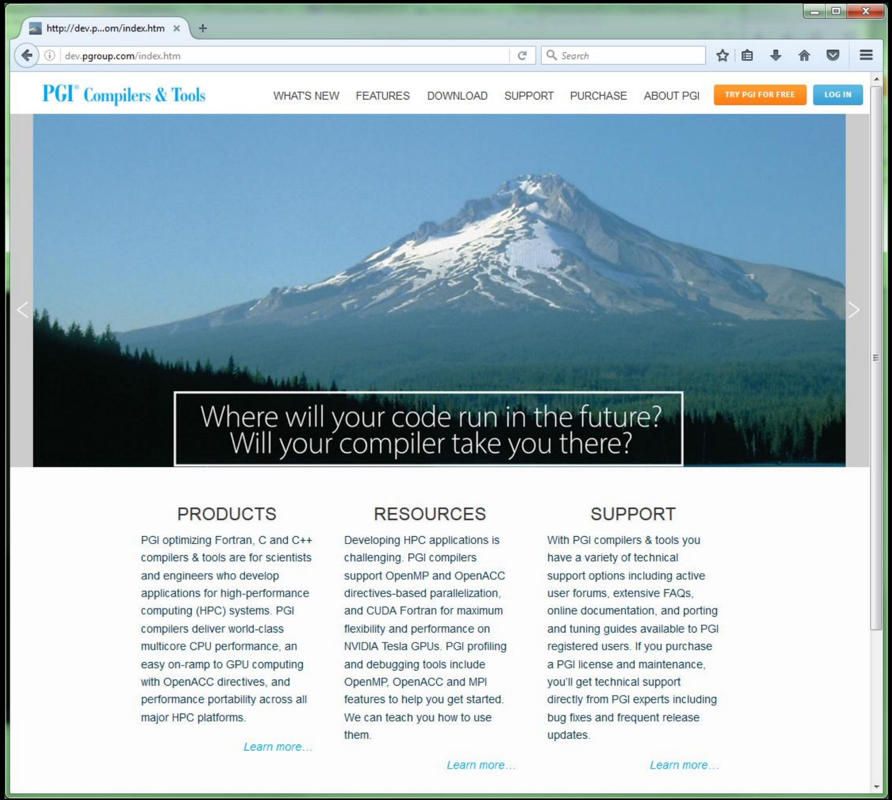
Supported on Linux, macOS, Windows

MPI/OpenMP/OpenACC Tools

PGDBG® debugger

PGPROF® profiler

Interoperable with DDT, Totalview



www.pgroup.com

Riding Waves of Disruption

PGI Founded
Jan 3, 1989

TFLOPs / Beowulf

x86-64 vs IA64

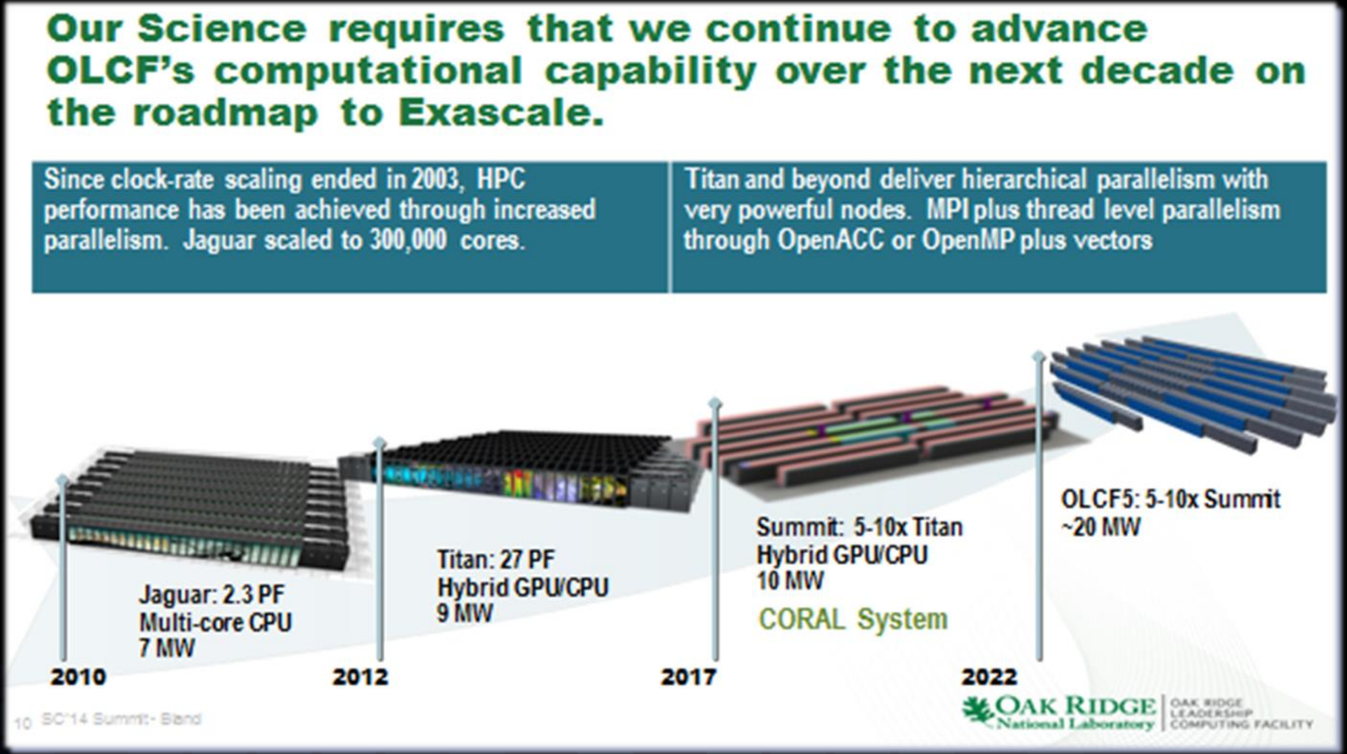
Accelerated Computing

OpenPOWER™

PGI + LLVM

1989 | 1997 | 2003 | 2009 | 2014

ORNL Leadership Computing Systems, CORAL



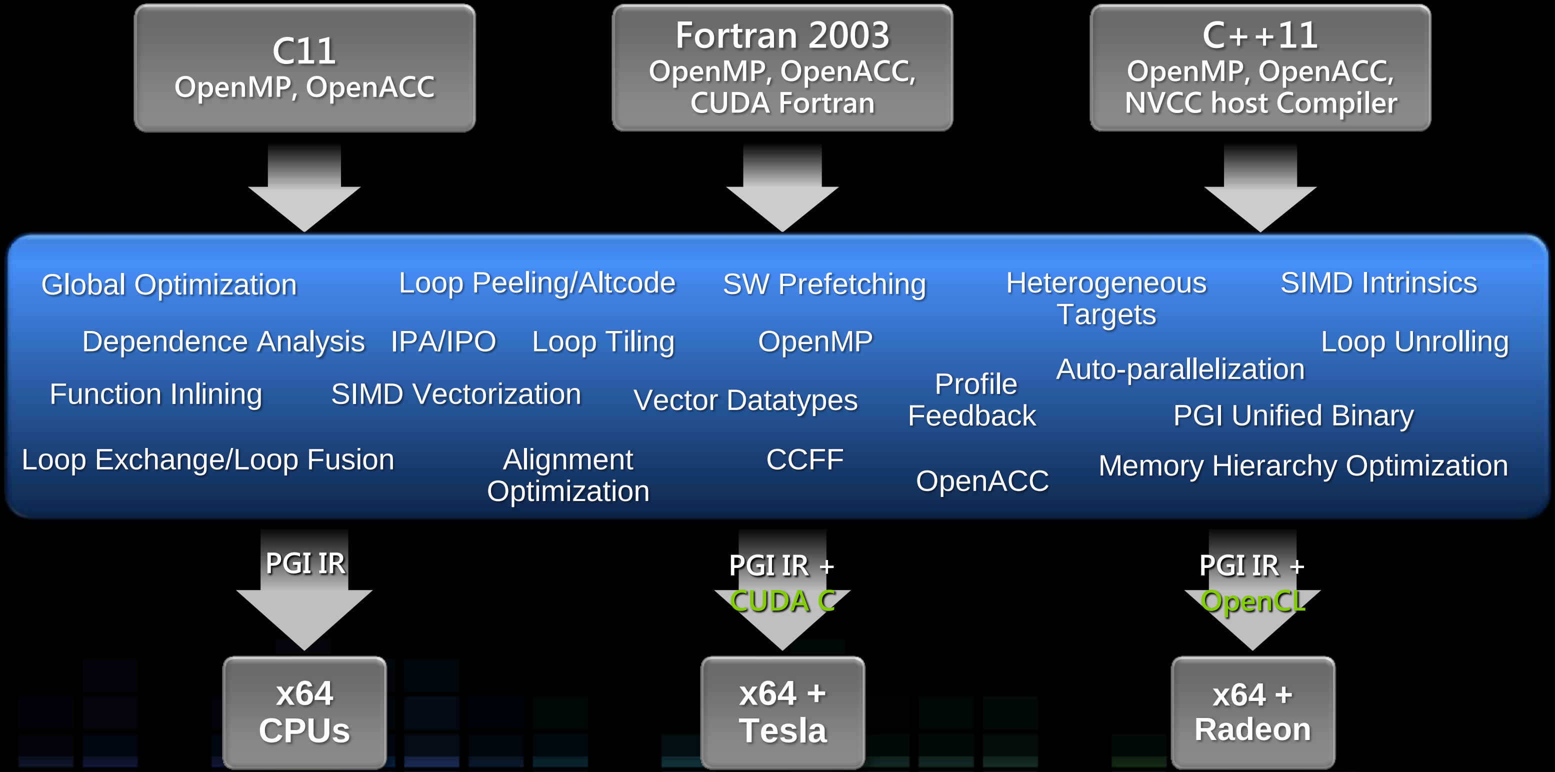
PGI®

PGI and CORAL

" Porting and optimizing production HPC applications from one platform to another can be one of the most significant costs in the adoption of breakthrough hardware technologies. The PGI compiler has been our primary compiler on Jaguar and Titan since 2005. Having the PGI compiler suite available in the POWER environment will provide continuity and facilitate code portability of existing CPU-only and GPU-enabled Titan applications to our next major system. "

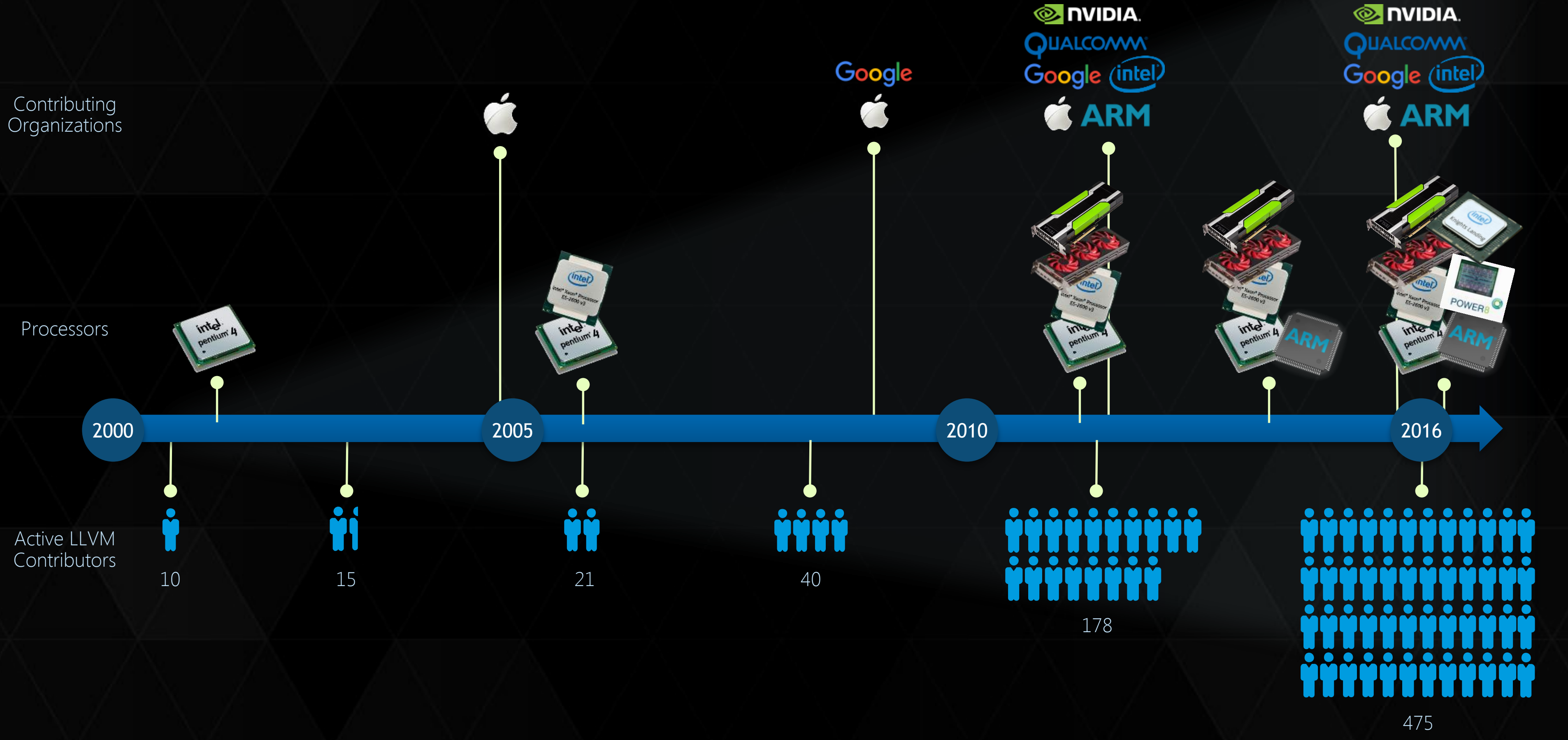
— *Buddy Bland, Titan Project Director, Oak Ridge National Lab*

PGI Compilers 2014 …

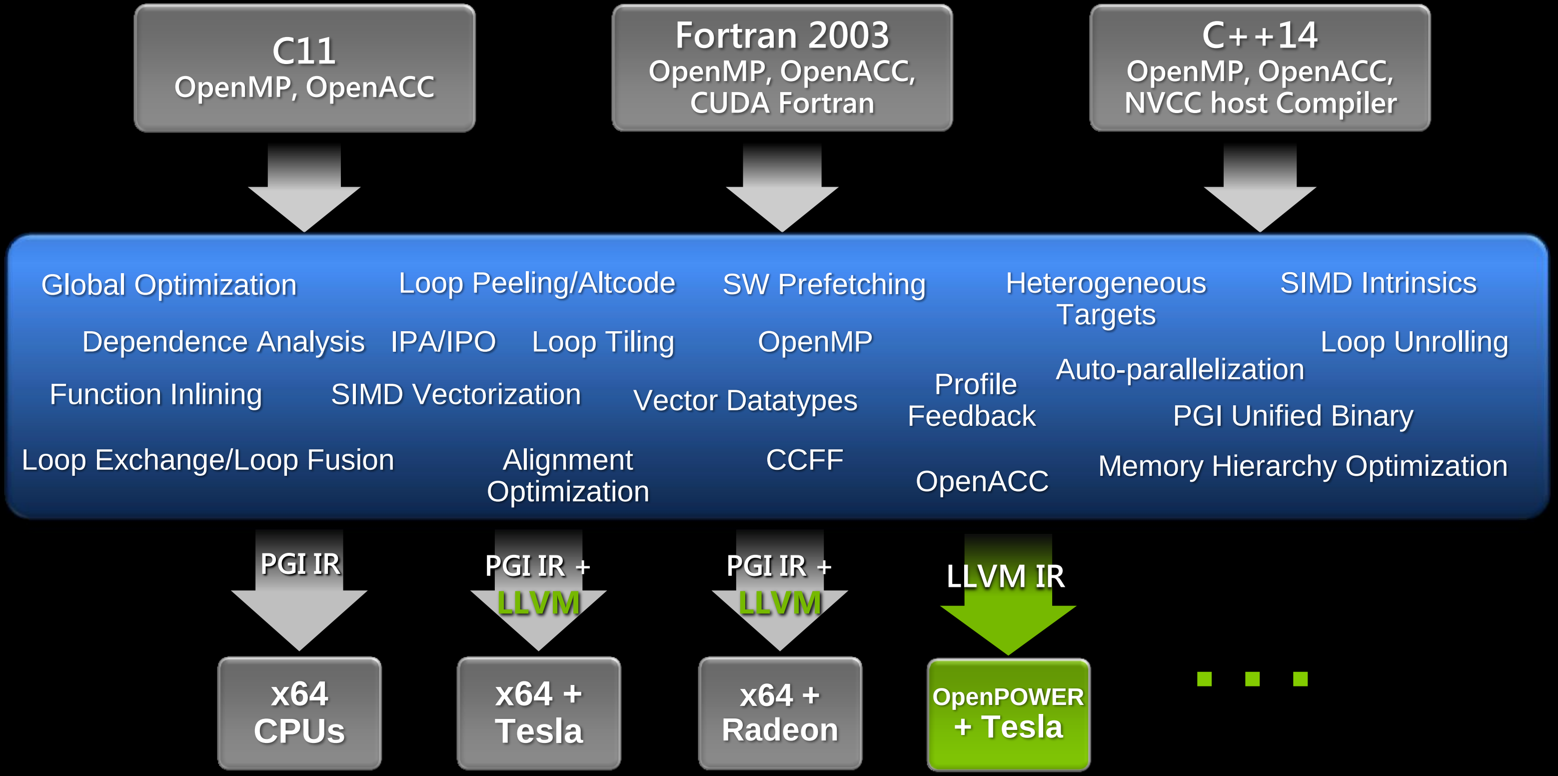


PGI®

LLVM: Community Power



PGI Compilers 2016 ...



PGI®

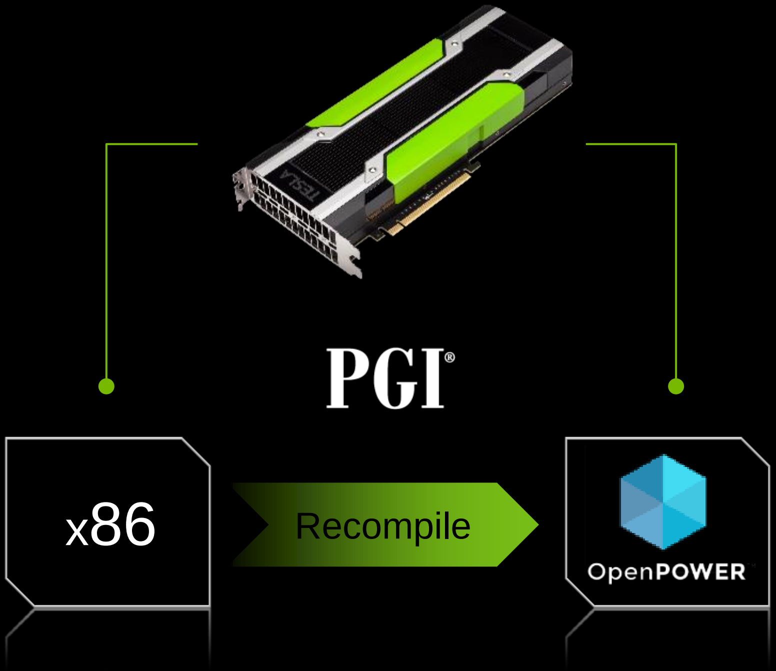
PGI for OpenPOWER+Tesla

Fortran 2003, C11, C++14 compilers, PGPROF profiler

CUDA Fortran, OpenACC, OpenMP, NVCC host compiler

Integrated with LLVM for OpenPOWER code generation

First production release now available



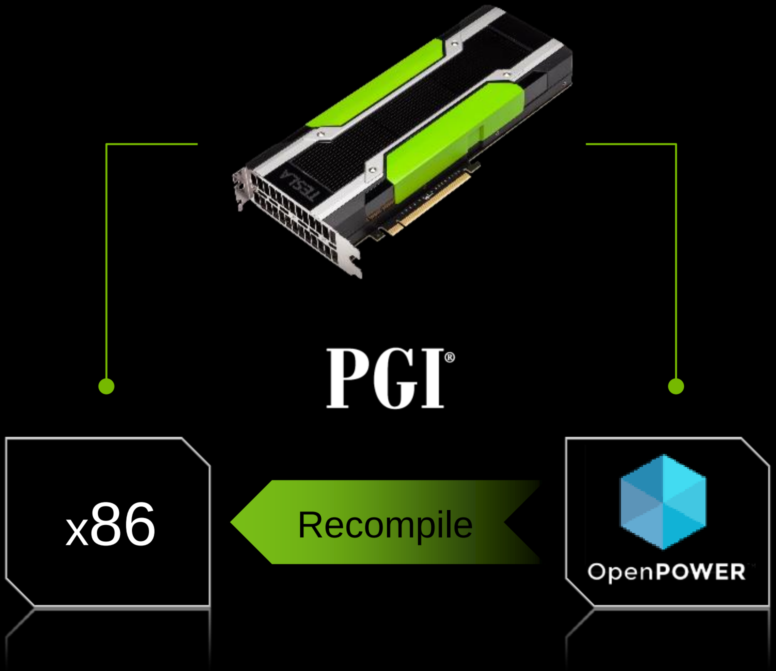
PGI for OpenPOWER+Tesla

Fortran 2003, C11, C++14 compilers, PGPROF profiler

CUDA Fortran, OpenACC, OpenMP, NVCC host compiler

Integrated with LLVM for OpenPOWER code generation

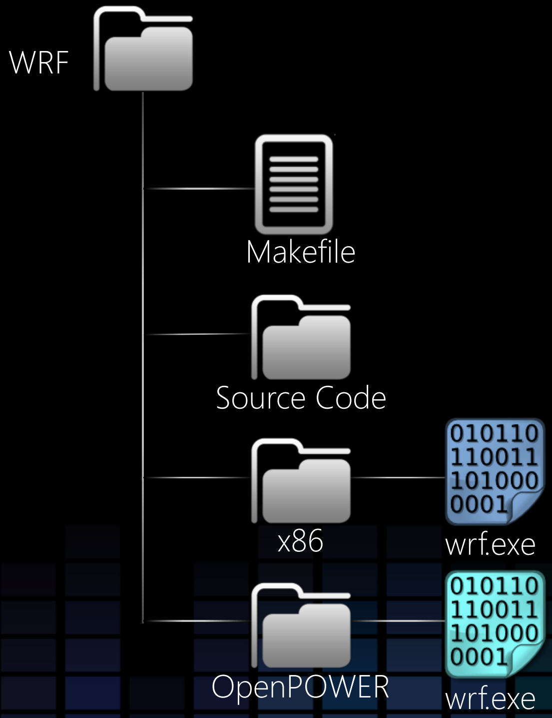
First production release now available



PGI

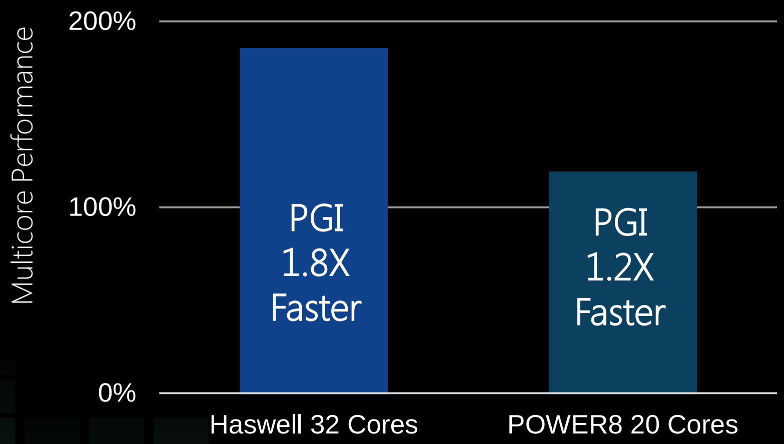
Porting an 800K line HPC Application from x86 to OpenPOWER

Recompile ...



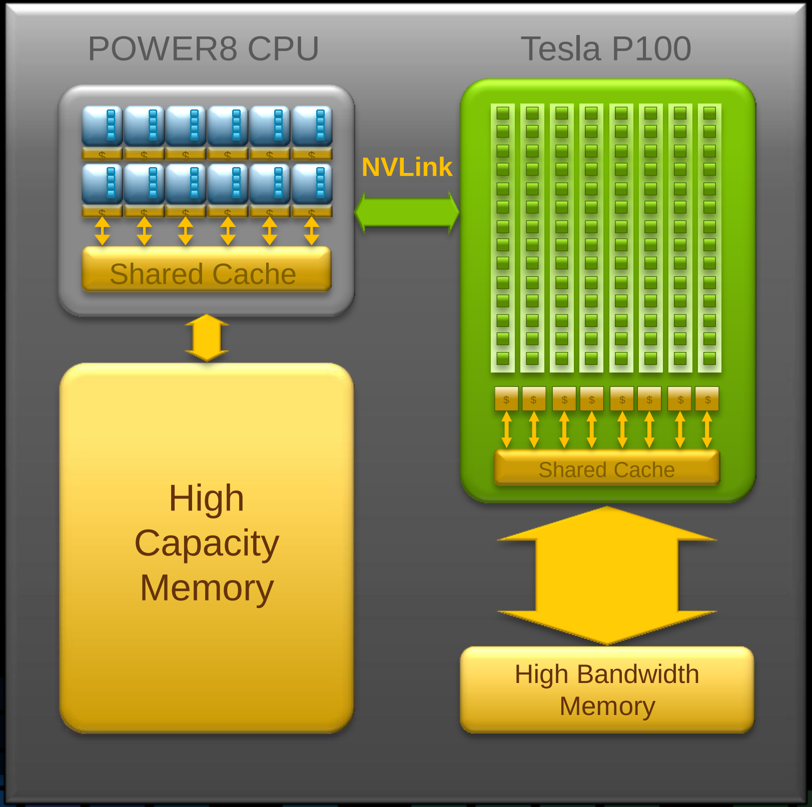
Run ...

WRF 3.8.1 OpenMP Performance
PGI 16.10 vs GNU 6.1



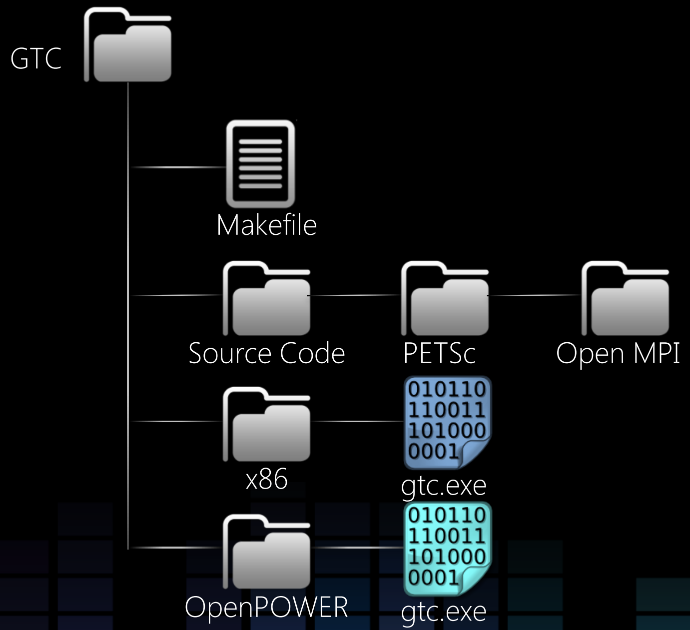
x86 CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores
OpenPOWER CPU: IBM 8247-42L POWER8E, 4 sockets, 20 cores
PGI options: -fast -Mstack_arrays -mp
GNU options: -O3 -funroll-loops -fpeel-loops -fopenmp

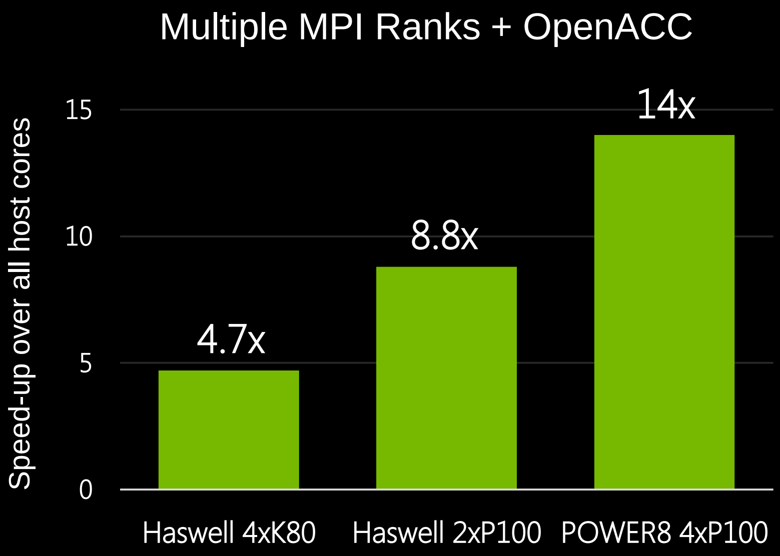
OpenPOWER+Tesla HPC Node



PGI®

Porting the Gyrokinetic Toroidal Code (GTC) from Xeon+Tesla to OpenPOWER+Tesla using OpenACC





X86 CPU: Intel Xeon E5-2698 v3,
POWER CPU: IBM POWER8NVL

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

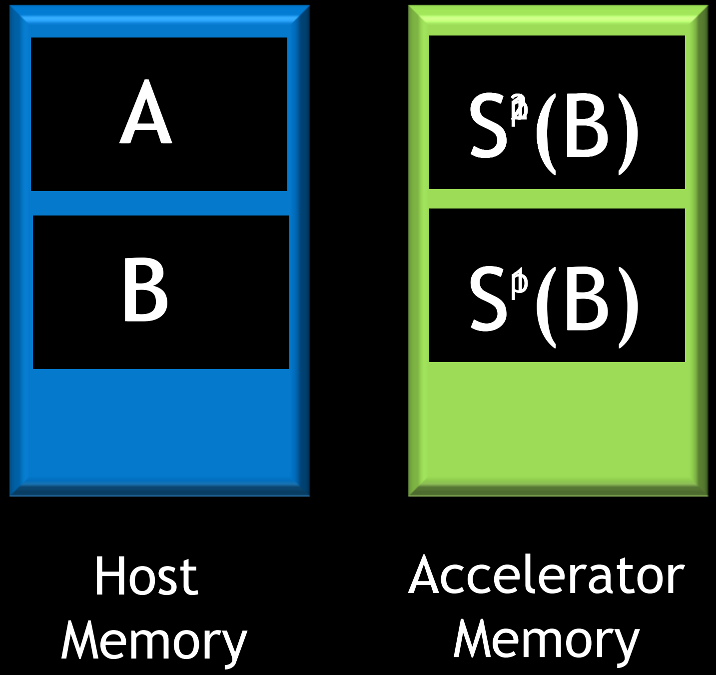
```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          c[i] = a[i] + b[i];
          ...
      }
  }
  ...
}
```

OpenACC
Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

OpenACC in a Nutshell

```
...
#pragma acc data copy(b[0:n][0:m]) \
                 create(a[0:n][0:m])
{
for (iter = 1; iter <= p; ++iter){
  #pragma acc kernels
  {
  for (i = 1; i < n-1; ++i){
    for (j = 1; j < m-1; ++j){
      a[i][j]=w0*b[i][j]+
              w1*(b[i-1][j]+b[i+1][j]+
                  b[i][j-1]+b[i][j+1])+
              w2*(b[i-1][j-1]+b[i-1][j+1]+
                  b[i+1][j-1]+b[i+1][j+1]);
  } }
  for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
      b[i][j] = a[i][j];
  }
}
}
...
```



OpenACC for Multicore CPUs & GPUs

```
!$acc kernels loop
 do j = 1, m
    do i = 1, n
       a(j,i) = b(j,i)*alpha + c(i,j)*beta
    enddo
 enddo
```

CPU

```
% pgfortran a.f90 -ta=multicore -c -Minfo
sub:
  10, Loop is parallelizable
      Generating Multicore code
      10, !$acc loop gang
  11, Loop is parallelizable
```

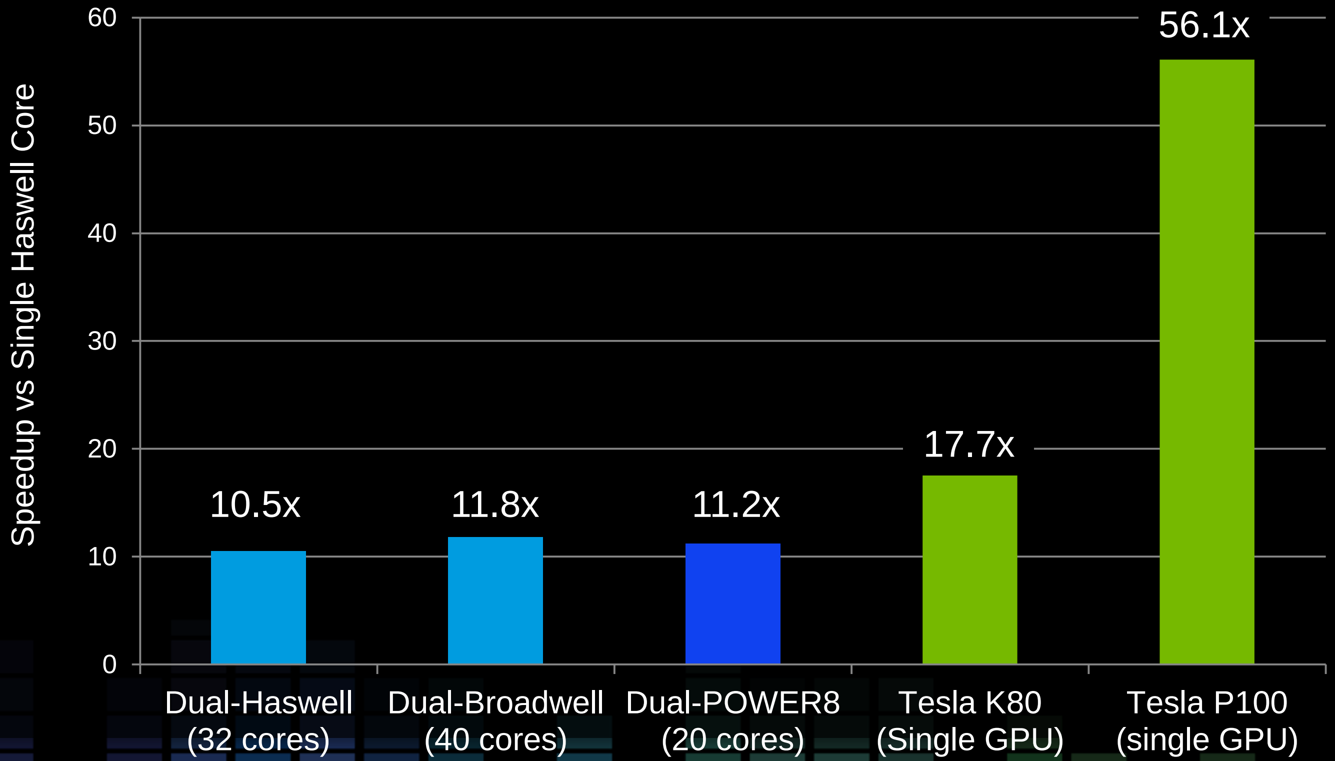
GPU

```
% pgfortran a.f90 -ta=tesla -c -Minfo
sub:
  10, Loop is parallelizable
  11, Loop is parallelizable
      Accelerator kernel generated
      Generating Tesla code
      10, !$acc loop gang, vector(4)
      11, !$acc loop gang, vector(32)
```

PGI

PGI OpenACC - SPEC ACCEL 1.0 Benchmarks

Geometric mean across all 15 benchmarks



PGI®

Performance measured November, 2016 and are considered estimates per SPEC run and reporting rules. SPEC® and SPEC ACCEL® are registered trademarks of the Standard Performance Evaluation Corporation (www.spec.org).

CUDA Fortran for Tesla GPUs

```
real, device, allocatable, dimension(:,:) ::
        Adev,Bdev,Cdev

. . .

allocate (Adev(N,M), Bdev(M,L), Cdev(N,L))
Adev = A(1:N,1:M)
Bdev = B(1:M,1:L)

call mm_kernel <<<dim3(N/16,M/16),dim3(16,16)>>>
              ( Adev, Bdev, Cdev, N, M, L )

C(1:N,1:L) = Cdev
deallocate ( Adev, Bdev, Cdev )

. . .
```

```
attributes(global) subroutine mm_kernel
            ( A, B, C, N, M, L )
real :: A(N,M), B(M,L), C(N,L), Cij
integer, value :: N, M, L
integer :: i, j, kb, k, tx, ty
real, shared :: Asub(16,16),Bsub(16,16)
tx = threadidx%x
ty = threadidx%y
i = blockidx%x * 16 + tx
j = blockidx%y * 16 + ty
Cij = 0.0
do kb = 1, M, 16
   Asub(tx,ty) = A(i,kb+tx-1)
   Bsub(tx,ty) = B(kb+ty-1,j)
   call syncthreads()
   do k = 1,16
      Cij = Cij + Asub(tx,k) * Bsub(k,ty)
   enddo
   call syncthreads()
enddo
C(i,j) = Cij
end subroutine mmul_kernel
```

CPU Code

Tesla Code

PGI®

!$CUF kernel directives

```
module madd_device_module
  use cudafor
contains
  subroutine madd_dev(a,b,c,sum,n1,n2)
    real,dimension(:,:),device :: a,b,c
    real :: sum
    integer :: n1,n2
    type(dim3) :: grid, block
!$cuf kernel do (2) <<<(*,*),(32,4)>>>
    do j = 1,n2
      do i = 1,n1
        a(i,j) = b(i,j) + c(i,j)
        sum = sum + a(i,j)
      enddo
    enddo
  end subroutine
end module
```

Equivalent hand-written CUDA kernels

```
module madd_device_module
  use cudafor
  implicit none
contains
  attributes(global) subroutine madd_kernel(a,b,c,blocksum,n1,n2)
    real, dimension(:,:) :: a,b,c
    real, dimension(:) :: blocksum
    integer, value :: n1,n2
    integer :: i,j,tindex,tneighbor,bindex
    real :: mysum
    real, shared :: bsum(256)
! Do this thread's work
    mysum = 0.0
    do j = threadidx%y + (blockidx%y-1)*blockdim%y, n2, blockdim%y*griddim%y
      do i = threadidx%x + (blockidx%x-1)*blockdim%x, n1, blockdim%x*griddim%x
        a(i,j) = b(i,j) + c(i,j)
        mysum = mysum + a(i,j) ! accumulates partial sum per thread
      enddo
    enddo
! Now add up all partial sums for the whole thread block
! Compute this thread's linear index in the thread block
! We assume 256 threads in the thread block
    tindex = threadidx%x + (threadidx%y-1)*blockdim%x
! Store this thread's partial sum in the shared memory block
    bsum(tindex) = mysum
    call syncthreads()
! Accumulate all the partial sums for this thread block to a single value
    tneighbor = 128
    do while( tneighbor >= 1 )
      if( tindex <= tneighbor ) &
        bsum(tindex) = bsum(tindex) + bsum(tindex+tneighbor)
      tneighbor = tneighbor / 2
      call syncthreads()
    enddo
! Store the partial sum for the thread block
    bindex = blockidx%x + (blockidx%y-1)*griddim%x
    if( tindex == 1 ) blocksum(bindex) = bsum(1)
  end subroutine

! Add up partial sums for all thread blocks to a single cumulative sum
  attributes(global) subroutine madd_sum_kernel(blocksum,dsum,nb)
    real, dimension(:) :: blocksum
    real :: dsum
    integer, value :: nb
    real, shared :: bsum(256)
    integer :: tindex,tneighbor,i
! Again, we assume 256 threads in the thread block
! accumulate a partial sum for each thread
    tindex = threadidx%x
    bsum(tindex) = 0.0
    do i = tindex, nb, blockdim%x
      bsum(tindex) = bsum(tindex) + blocksum(i)
    enddo
    call syncthreads()
! This code is copied from the previous kernel
! Accumulate all the partial sums for this thread block to a single value
! Since there is only one thread block, this single value is the final result
    tneighbor = 128
    do while( tneighbor >= 1 )
      if( tindex <= tneighbor ) &
        bsum(tindex) = bsum(tindex) + bsum(tindex+tneighbor)
      tneighbor = tneighbor / 2
      call syncthreads()
    enddo
    if( tindex == 1 ) dsum = bsum(1)
  end subroutine

  subroutine madd_dev(a,b,c,dsum,n1,n2)
    real, dimension(:,:), device :: a,b,c
    real, device :: dsum
    real, dimension(:), allocatable, device :: blocksum
    integer :: n1,n2,nb
    type(dim3) :: grid, block
    integer :: r
! Compute grid/block size; block size must be 256 threads
    grid = dim3((n1+31)/32, (n2+7)/8, 1)
    block = dim3(32,8,1)
    nb = grid%x * grid%y
    allocate(blocksum(1:nb))
    call madd_kernel<<< grid, block >>>(a,b,c,blocksum,n1,n2)
    call madd_sum_kernel<<< 1, 256 >>>(blocksum,dsum,nb)
    r = cudaThreadSynchronize() ! don't deallocate too early
    deallocate(blocksum)
  end subroutine
end module
```

PGI®

PGI + LLVM Integration

PGI Compilers 2016 …

C11
OpenMP, OpenACC

Fortran 2003
OpenMP, OpenACC,
CUDA Fortran

C++14
OpenMP, OpenACC,
NVCC host Compiler

Global Optimization
Loop Peeling/Altcode
SW Prefetching
Heterogeneous Targets
SIMD Intrinsics
Dependence Analysis
IPA/IPO
Loop Tiling
OpenMP
Loop Unrolling
Auto-parallelization
Function Inlining
SIMD Vectorization
Vector Datatypes
Profile Feedback
PGI Unified Binary
Loop Exchange/Loop Fusion
Alignment Optimization
CCFF
OpenACC
Memory Hierarchy Optimization

PGI IR → **x64 CPUs**

PGI IR + LLVM → **x64 + Tesla**

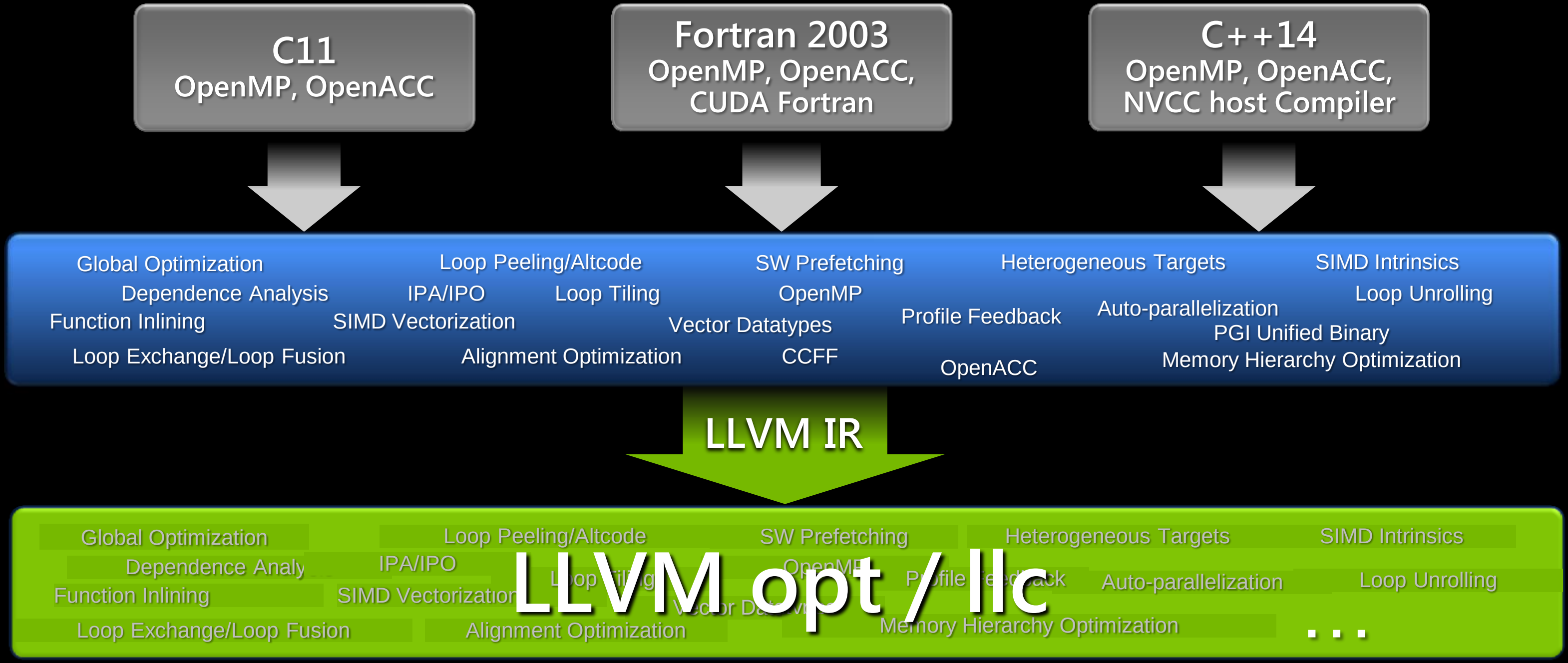
PGI IR + LLVM → **x64 + Radeon**

LLVM IR → **OpenPOWER + Tesla**

…

PGI®

LLVM is not a code generator …



it is "… a collection of modular and reusable compiler and toolchain technologies."

PGI®

Integrating LLVM into the PGI Compilers

- PGI ILI -> LLVM IR bridge, CPU-side and GPU-side
- C/C++/Fortran language support, scalar code generation
- Target independent vectorizer
- OpenMP re-implementation, SMP auto-parallelization
- Enabling OpenACC and CUDA Fortran
- Integration, Testing, Documentation
- Dovetailing PGI optimizer and LLVM opt

Target-independent vectorizer

```fortran
subroutine add_exp(n,a,b,c)
  integer n
  real*8, dimension(n) :: a,b,c
  integer i

  do i = 1,n
    a(i) = b(i) + exp(c(i))
  enddo

end subroutine
```

```llvm
L.LB1_428:
    . . .
    %14 = getelementptr i8, i8* %11, i64 %13       # compute c(i) location
    %15 = bitcast i8* %14 to <4 x double>*         # cast c(i) as vector of 4 doubles
    %16 = load <4 x double>, <4 x double>* %15, align 8
    %17 = bitcast <4 x double> (...)* @__gvd_exp4 to <4 x double> (<4 x double>)*
    %18 = call <4 x double> %17 (<4 x double> %16) # call vector exp on 4 doubles
    . . .
    %22 = getelementptr i8, i8* %19, i64 %21       # compute b(i) location
    %23 = bitcast i8* %22 to <4 x double>*         # cast b(i) as vector of 4 doubles
    %24 = load <4 x double>, <4 x double>* %23, align 8
    %25 = fadd <4 x double> %18, %24               # add b(i) to exp(c(i))
    . . .
    %28 = getelementptr i8, i8* %26, i64 %27       # compute a(i) location
    %29 = bitcast i8* %28 to <4 x double>*         # cast a(i) as vector of 4 doubles
    store <4 x double> %25, <4 x double>* %29, align 1 # store result to a(i)
    . . .
    br i1 %33, label %L.LB1_428, label %L.LB1_473  # loop
    …
    declare <4 x double> @__gvd_exp4(...)
```

Leveraging LLVM Vector Data Types

PGI

Target-independent vectorizer

```fortran
subroutine add_exp(n,a,b,c)
  integer n
  real*8, dimension(n) :: a,b,c
  integer i

  do i = 1,n
    a(i) = b(i) + exp(c(i))
  enddo

end subroutine
```

```
    leal    -3(%rbx), %r12d
.LBB0_3:
    vmovupd (%r14,%rbp), %ymm0
    callq   __gvd_exp4
    vaddpd  (%r15,%rbp), %ymm0, %ymm0
    vmovupd %ymm0, (%r13,%rbp)
    addq    $32, %rbp
    addl    $-4, %r12d
    testl   %r12d, %r12d
    jg  .LBB0_3
```

x86-64 AVX-256

```
.LBB0_4:
   lxvd2x 0, 30, 24
   stxvd2x 0, 1, 23
   ori 2, 2, 0
   lxvd2x 0, 1, 23
   ld 4, 40(1)
   ld 3, 32(1)
   xxswapd  34, 0
   bl __gvd_exp2
   nop
   lxvd2x 0, 29, 24
   addi 22, 22, -2
   cmpwi    22, 0
   xxswapd  0, 0
   xvadddp 0, 34, 0
   xxswapd  0, 0
   stxvd2x 0, 28, 24
   addi 24, 24, 16
   bgt  0, .LBB0_4
```

OpenPOWER VSX

Outlining parallel regions

```
subroutine add_exp(n,a,b,c)
  integer n
  real*8, dimension(n) :: a,b,c
  integer i
!$omp parallel do
  do i = 1,n
    a(i) = b(i) + exp(c(i))
  enddo

end subroutine
```

```
##  lineno: 5
..LN1:
        movq    .STATICS1(%rip), %rdi
        movl    $2, %esi
        vzeroupper
        .p2align        4,,1
        call    _mp_penter
        vzeroupper
        .p2align        4,,1
        call    _mp_lcpu
        movl    %eax, 268(%rsp)
        vzeroupper
        .p2align        4,,1
        call    _mp_ncpus
        movl    268(%rsp), %ecx
        . . .
        # Execute SIMD vector loop
        # in parallel
        . . .
        call    _mp_pexit
        movq    -72(%rbp), %r15
        movq    -64(%rbp), %r14
```

PGI Native Inline Parallel Regions

```
. . .
xorl    %edi, %edi
callq   __kmpc_global_thread_num
movl    (%rbx), %eax
movl    %eax, 4(%rsp)
movq    %rbx, 16(%rsp)
movq    %r12, 24(%rsp)
leaq    4(%rsp), %rax
movq    %rax, 32(%rsp)
movq    %r15, 40(%rsp)
movq    %r14, 48(%rsp)
leaq    8(%rsp), %rcx
movl    $0, %edi
movl    $1, %esi
movl    $add_exp__1F1L5_, %edx
xorl    %eax, %eax
callq   __kmpc_fork_call
addq    $56, %rsp
popq    %rbx
popq    %r12
popq    %r14
popq    %r15
retq
```

Parallel Regions Outlined for LLVM

PGI®

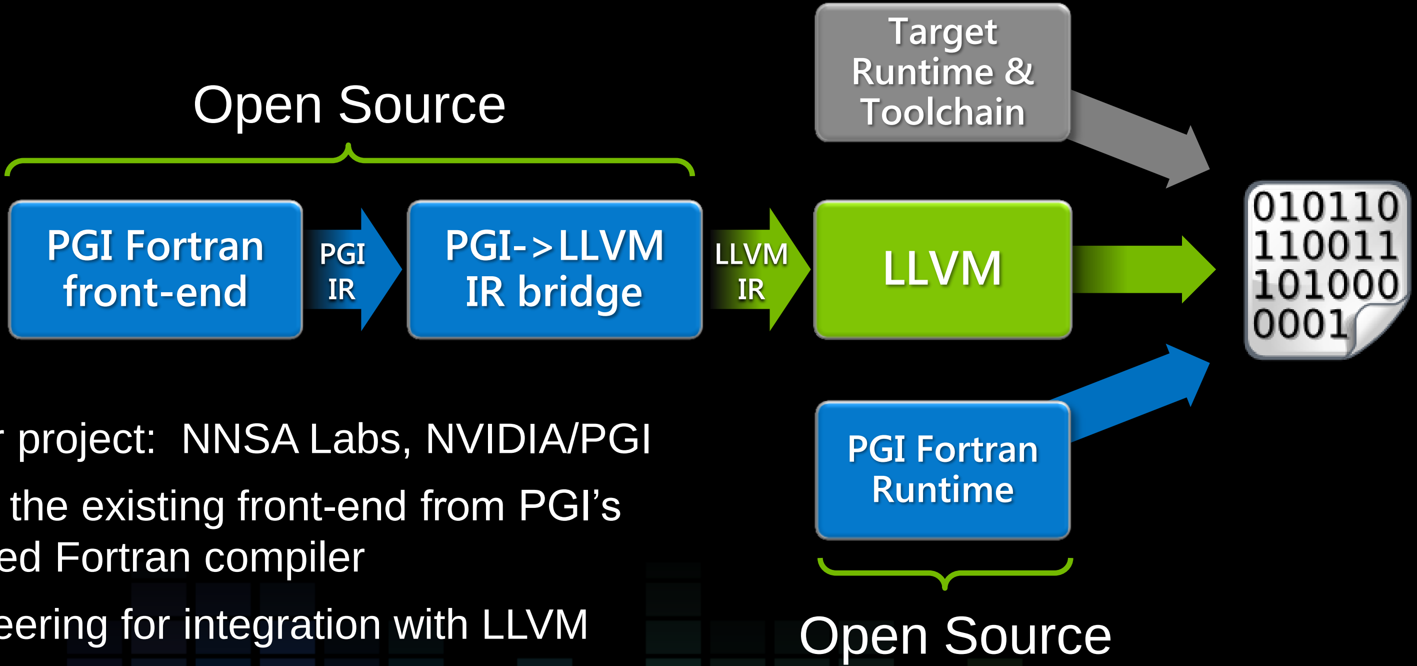
PGI + LLVM to do list

- Fortran DWARF generation
- OpenPOWER performance analysis
- OpenMP performance tuning, OpenMP 4.5
- PGI vectorizer performance tuning
- Dovetailing PGI optimizer and LLVM opt
- POWER9 128-bit IEEE floating-point support

Flang

An open source Fortran front-end for LLVM

a.k.a. the Flang project

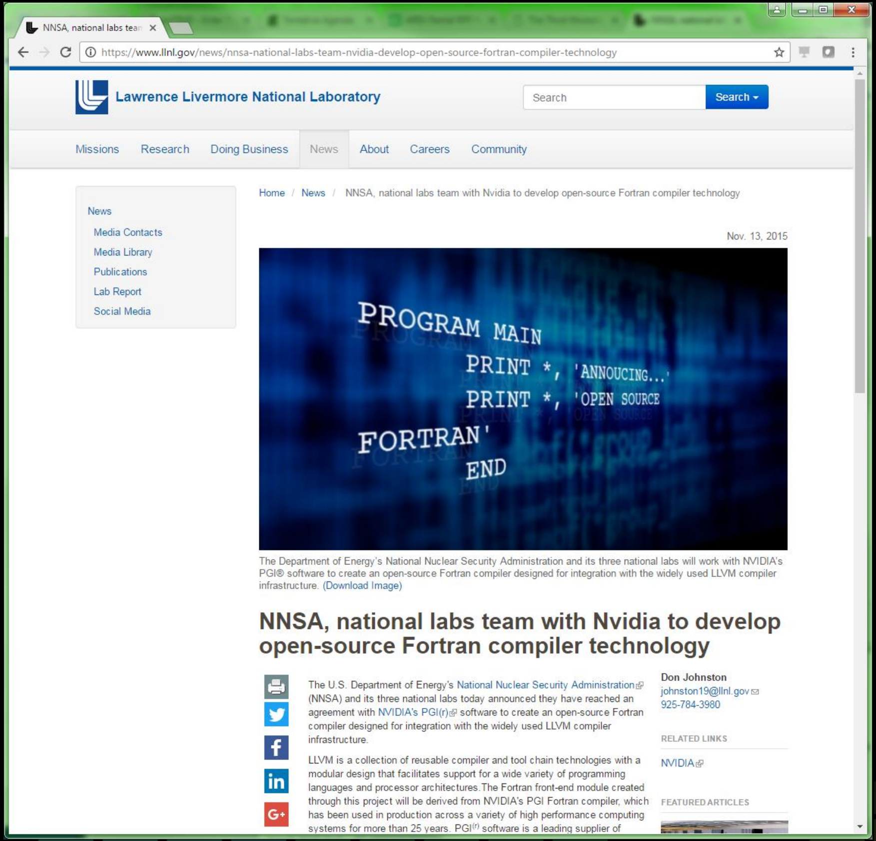


- Multi-year project: NNSA Labs, NVIDIA/PGI
- Based on the existing front-end from PGI's widely-used Fortran compiler
- Re-engineering for integration with LLVM
- Develop CLANG-quality Fortran msg facility

Many Stakeholders, Many Goals

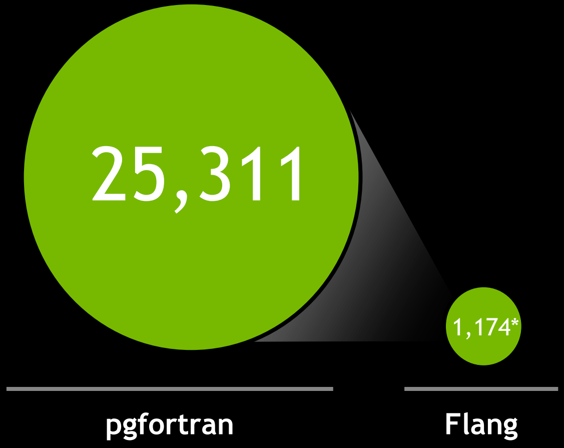
LANL	New developer productive in source base in 4 – 8 weeks
Sandia	Single-thread/SIMD and OpenMP 3.1 performance
LLNL	OpenMP 4.x features, GPU and OpenPOWER support
NVIDIA	Accelerate Fortran features support, PGI interoperability
Everyone	Adoption by both the HPC and LLVM communities

ANL, IBM, ARM Ltd, ORNL, Codethink, …



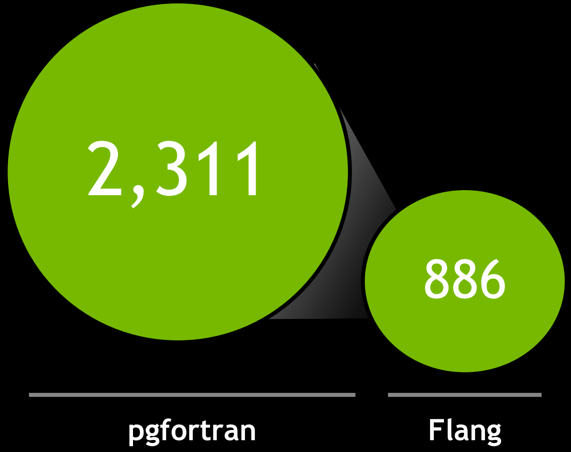
Creating the initial Flang source base

- Front-end 85%
- Runtime Libraries 15%



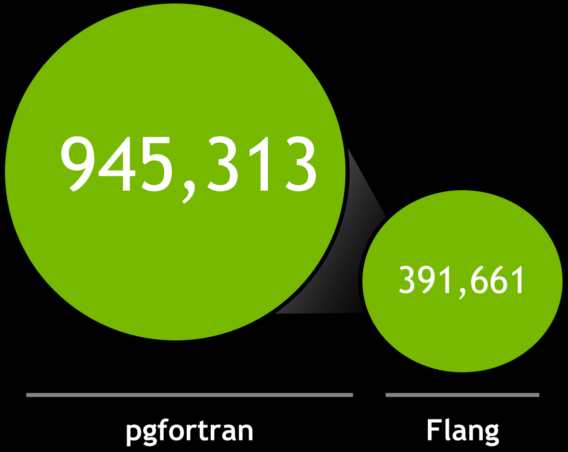
95% fewer #ifdefs

- Front-end 85%
- Runtime Libraries 15%



62% fewer files

- Front-end 80%
- Runtime Libraries 20%



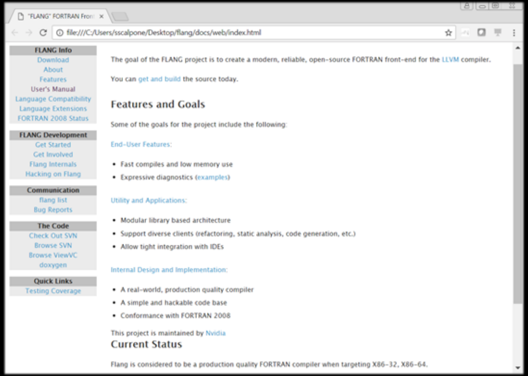
59% fewer LOC

*Clang has 212 #ifdefs in lib, include, tools

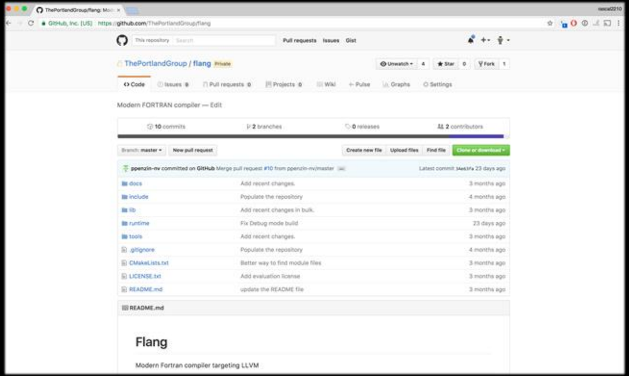
Flang Development Status

- Source code clean-up, refactoring & documentation ongoing
- Vendor neutrality nearly complete
- Frequent source and Flang binary updates to partners
- Passes most PGI Fortran Linux/x86 QA tests
- SIMD vectorization via the LLVM vectorizer, tuning ongoing
- Most of OpenMP 4.5 is implemented (CPU-side only)

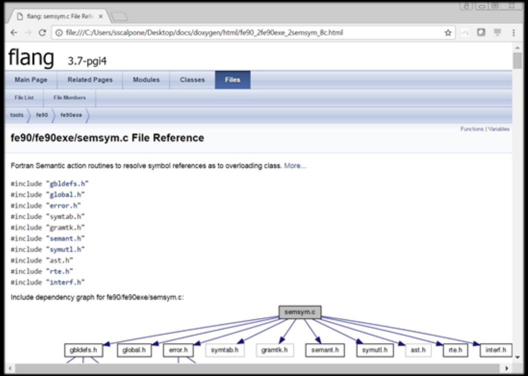
Flang Source Code



Home page



Github



Doxygen

Flang Single-core Performance

SPEC CPU 2006 Fortran codes, all times in seconds, 1 Haswell core

	PGI FORTRAN 16.10	GFORTRAN 6.1	FLANG DEV LLVM 3.9
410.bwaves	182s	220s	251s
416.gamess	507s	Fails	475s
434.zeusmp	183s	221s	240s
436.cactusADM	165s	194s	208s
437.leslie3d	179s	209s	435s
454.calculix	171s	297s	608s
459.GemsFDTD	261s	286s	391s
465.tonto	295s	373s	Fails
481.wrf	157s	271s	247s

PGI Fortran: -fast -Mfprelaxed -Mstack_arrays gfortran: -O3 -funroll-loops -fpeel-loops -ffast-math Flang: -O3 -march=core-avx2 -ffp-contract=fast -Knoieee
Performance measured November, 2016 and are considered estimates per SPEC run and reporting rules. SPEC® and SPEC CPU® are registered trademarks of the Standard Performance Evaluation Corporation (www.spec.org).

Flang OpenMP Performance

SPEC OMP 2012 Fortran codes, all times in seconds, 32 Haswell cores (64 threads)

	PGI FORTRAN 16.10	GFORTRAN 6.1	FLANG DEV LLVM 3.9
350.md	517s	3460s	459s
351.bwaves	469s	519s	805s
357.bt331	449s	492s	474s
360.ilbdc	541s	6846s	539s
362.fma3d	575s	504s	656s
363.swim	633s	634s	632s
370.mgrid	693s	697s	690s
371.applu	451s	414s	514s

PGI Fortran: -fast -mp -Mfprelaxed -Mstack_arrays gfortran: -O3 -funroll-loops -fpeel-loops -ffast-math –fopenmp
Flang: -O3 -mp -march=core-avx2 -ffp-contract=fast -Knoieee All: OMP_NUM_THREADS=64 OMP_PROC_BIND=true
Performance measured November, 2016 and are considered estimates per SPEC run and reporting rules. SPEC® and SPEC OMP®
are registered trademarks of the Standard Performance Evaluation Corporation (www.spec.org).

Flang Year 2 Development Plans

- Source code
  - Continue source clean-up, refactoring, documentation
  - Create repository and release as open source
  - Deploy an open source testing infrastructure
- Features
  - Enhance compile-time Fortran error/warning messages
  - Incremental F08 and OpenMP 4.5 features
  - LLVM enhancements to enable Fortran DWARF generation
- Performance
  - Incremental, likely to be reactive after initial pass is done

Concluding Thoughts

- LLVM is integral to HPC compilers at NVIDIA and PGI
- Fortran ➞ First-class citizen in the LLVM community
- LLVM as a platform for out-of-tree developers