Architect of an Open World™

Le compilateur, un maillon essentiel de la performance des codes.

Lyon – 16 Avril 2013

Gunter Roeth
Applications Engineer
Applications & Performance Team
Extreme Computing Business Unit

PLAN

1. COMPILATEURS POUR LE CALCUL HAUTE PERFORMANCE
2. PHASES D'OPTIMISATION
3. AVX INSTRUCTION SET (VECTORISATION)
4. COMPILATEURS INTEL
5. MODÈLES DE PROGRAMMATION DES XEON PHI
6. COMPILATEURS GNU
7. COMPILATEURS PGI
8. MODÈLES DE PROGRAMMATION DES GPUS
9. COMPILATEUR HMPP CAPS

COMPILERS OPTIMIZATIONS PASSES

- Local Optimization on basic blocks
  sequence of branchless statements in which the control flow enters and leaves at the end
    - algebraic identity removal, constant folding,
    - common sub-expression elimination, redundant load and store elimination,
    - scheduling, strength reduction, peephole optimizations.

- Global Optimization
  program unit over all basic blocks.
    - control-flow and data-flow analysis,
    - all loops are detected
    - constant propagation, copy propagation, dead store elimination,
    - global register allocation, invariant code motion, induction variable elimination

- Loop Optimization
    - unrolling, blocking, peeling, vectorization, and parallelization
- Function Inlining
    - increase code size and generate less efficient code

COMPILERS OPTIMIZATIONS PASSES

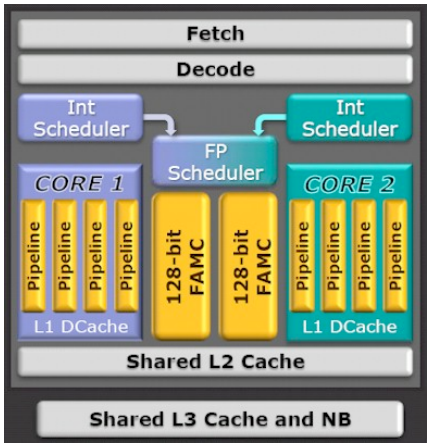
- Interprocedural Analysis (IPA) and Optimization IPO
  information across function call boundaries that would otherwise be unavailable
    - data alignment, argument removal, constant propagation, pointer disambiguation,
    - pure function detection, F90/F95 array shape propagation, C++ class hierarchy analysis,
    - data placement, inlining of functions from pre-compiled libraries
    - mod/ref analysis (may be modified/may be referenced), induction variable recognition,
    - routine key-attribute propagation, dead function elimination

- Profile-Feedback Optimization (PFO), profile guided (PGO) ,
  two-phase process: compilation and execution of an instrumented executable, re-compilation
    - branch frequency, function and subroutine call frequency, semi-invariant values,
    - loop index ranges, and other input data dependent information.
    - data set dependency

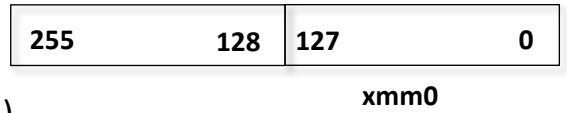
INTEL AND AMD SSE/AVX

2007: Intel (Woodcrest) and AMD (Barcelona) SSE 128bit (data paths and FP units)
2x speed-up of vectorized codes for DP using packed SSE instructions
Chips suffering from peak transfer rate vs peak FP performance (L1,L2,Memory)

2011: Intel (SandyBridge) and AMD (Bulldozer) AVX 256bit
again a possible 2x speed-up of vectorized codes …
Impressive : with even more cores, the peak transfer rate vs peak FP performance has slightly improved (additional ports, faster memory).
VEX prefix 256 SIMD support can run 3 or 4 operand syntax (compared to 2 for x86 ISA)
AVX needs vectorization



The 2x can not be reached for real applications.
From Westmere (Nehalem) to Sandybridge not all components scale 2x (especially the L2/L3 cache bandwidth)
LMBench. Expect a 1.2-1.4x !

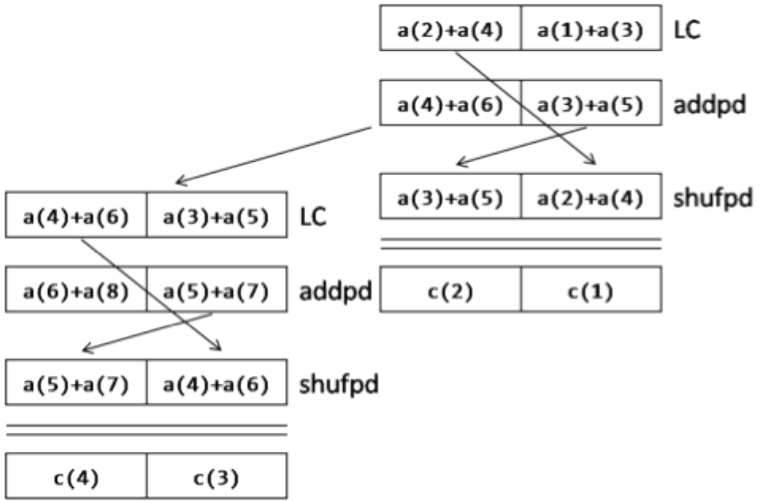


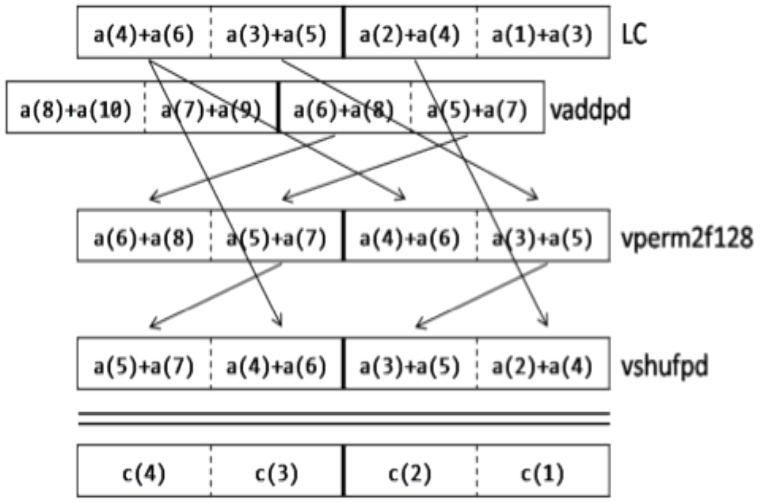
AMD shares FP unit for 2 cores. AMD extends FMA4 instructions (fused multiply-add).
AMD can run the threads using floating points in 256-bit AVX mode (scheduling ymm-based operations over the entire FP unit) or can use just one lane of the shared FP unit (xmm 128-bit) with VEX or SSE code.

XMM/YMM LOOPS COMPILER CHALLENGE

```
subroutine sum5(a, c, n)
real*8 a(n+4), c(n)
do i = 1, n
   c(i)=a(i)+a(i+1)+a(i+2)*2.d0
        +a(i+3)+a(i+4)
end do end
```

Using loop redundancy elimination (LRE) optimization, the compiler finds redundant expressions in the intermediate sums and carries those around to the next iteration (Loop Carried)

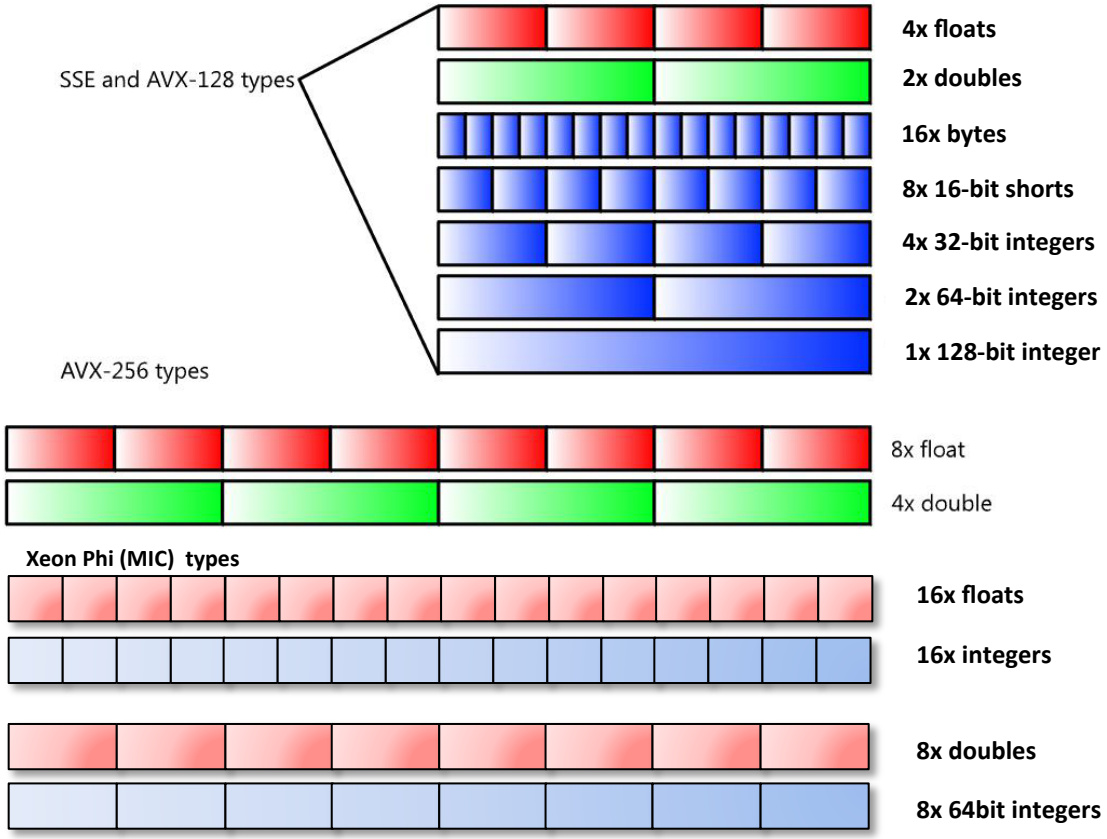




XMM : code requires one shuffle, but then produces two results for every three packed add operations.

YMM : AVX and the dual-lane give a 2x improvement, but the staging and manipulation increases in complexity.

INTEL SSE/AVX DATA TYPES



SSE

SSE-2

MMX™

Vector size: 64bit

Data types: 8, 16 and 32 bit ints

VL: 2,4,8

Intel® SSE

Vector size: 128bit

Data types: 8,16,32,64 bit ints
32 and 64bit floats

VL: 2,4,8,16

Intel® AVX

Vector size: 256bit

Data types:
32 and 64 bit floats

VL: 4, 8, 16

Intel® MIC

Vector size: 512bit

Data types: 32 and 64 bit ints
32 and 64 bit floats
(some support for 16 bits floats)

VL: 8,16

REQUIREMENTS FOR VECTORIZATION

- Must be a unit strided inner loop and may contain :
  - mathematical operators (sqrt, sin, exp,…)
  - if statements
  - reduction loops

- Avoid:
  - Function/subroutine calls (unless inlined)
  - Non-mathematical operators
  - Data-dependent loop exit conditions
  - Iteration count must be known at entry to loop
  - Loop carried data dependencies
  - Non-contiguous data (indirect addressing; non-unit stride)
  - Inefficient (compiler heuristics)
  - Align your data where possible
    - to 32 byte boundaries (for AVX instructions)
    - to 16 bytes, or at least to “natural” alignment

```
#pragma simd
for(int ray=0; ray < N; ray++) {
  float Color = 0.0f, Opacity = 0.0f;
  int len = 0;
  int upper = raylen[ray];
  while (len < upper) {
    int voxel = ray + len;
    len++;
    if(visible[voxel] == 0) continue;
    float O = opacity[voxel];
    if(O == 0.0) continue;
    float Shading = O + 1.0;
    Color += Shading * (1.0f -
Opacity);
    Opacity += O * (1.0f - Opacity);
    if(Opacity > THRESH) break;
  }
  color_out[ray] = Color;
}
```

INTEL FORTRAN COMPILERS

Intel Fortran can be traced back to the 1960s from the VAX Fortran on PDP-11 computers.

- The Intel Fortran Compiler for Linux is not binary compatible with gfortran compiler
- Many differences for accepted Fortran language/standard conformance
  - Fortran95, OMP, different name-mangling scheme

Intel Fortran Compiler for Linux is binary compatible with C-object files created with Intel or the GNU compiler.

Intel has a front/middle a shared back end.

Fortran 2003 implementation almost complete

Complete type-bound procedures (GENERIC, OPERATOR,..)
FINAL procedures
Bounds remapping on pointer assignments
Remaining features of F2003 :
User-defined derived type I/O
Parameterized derived types

Fortran 2008 features

Co-arrays
DO CONCURRENT
CONTIGUOUS
I/O enhancements
New constants in ISO_FORTRAN_ENV
New intrinsic functions
Increase maximum rank from 7 to 31

INTEL C AND C++ COMPILERS

Intel's C++ compiler XE 2013 is based on EDG (EDison Design Group)

- ANSI and ISO C and C++ standards
- most GNU C and C++ language extensions,
- OpenMP 3.0 standard
- excellent source, binary, and command-line compatibility with the GNU gcc3.4 and g++
  - Mixing and matching binary files created by g++, including third-party libraries
  - support the future C++ ABI (Application Binary Interface)

- Best is to use the Intel compiler for linking
  - **__intel_new_proc_init()** is added the main routinefrom libirc
  - compiler optimizations, such as OpenMP and vectorization

WAYS TO WRITE VECTOR CODE

Data Level Parallelism

(Auto-)Vectorization

```
for(i = 0; i < N; i++){
  A[i] = B[i] + C[i];
}
```

Cilk Array Notation for C/C++

```
A[:] = B[:] + C[:];
```

SIMD Pragma/Directive

```
#pragma simd
for(i = 0 i < N; i++) {
  A[i] = B[i] + C[i];
}
!DIR$ SIMD
do i = 1,N
  A(i) = B(i) + C(i)
enddo
```

Elemental Function

```
__declspec(vector)
float foo(float B, float C, int i)
{
  return B + C;
}
…
for(i = 0; i < N; i++) {
  A[i] = foo(B, C, i);
}
```

HLO OPTIMIZATION REPORTS

```
%ifort -O3 -opt_report_phase hlo -opt-report-phase hpo matmul.f90
HPO VECTORIZER REPORT (matmul_)
…
matmul.f90(9:1-9:1):VEC:matmul_: PERMUTED LOOP WAS VECTORIZED
…
High Level Optimizer Report (matmul_)
#of Array Refs Scalar Replaced in matmul_ at line 9=36
…
<matmul.f90;9:9;hlo_linear_trans;matmul_;0>
LOOP INTERCHANGE in loops at line: 9 8 7
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
…
<matmul.f90;9:9;hlo_unroll;matmul_;0>
Loop at line 9 blocked by 128
…
Loop at line 7 blocked by 128
Loop at line 8 blocked by 128
Loop at line 8 unrolled and jammed by 4
Loop at line 7 unrolled and jammed by 4
```

```
subroutine matmul(a,b,c,n)
real(8) a(n,n),b(n,n),c(n,n)
do j=1,n
   do i=1,n
      do k=1,n
         c(j,i)=c(j,i)+a(k,i)*b(j,k)
      enddo
   enddo
enddo
end
```

INTEL COMPILER REPORTS

Very detailed information of all work done by the compiler
a special vectorization report, can be piped into other scripts
-vec-report<n>

-opt-report-phase=

- ipo_inl
  - Interprocedural Optimization Inlining Report
- ilo
  - Intermediate Language Scalar Optimization
- hpo
  - High Performance Optimization
- hlo
  - High-level Optimization
- pgo
  - Profile Guided Optimizer

-guide

- get advice on how to help the compiler to vectorize loops

INTEL FLAGS

- **-O3** to do
  - loop transformations first
  - attention includes FP model fast=1 with „value-unsafe“ optimizations
  - -fp-model fast=2 -no-prec-div -no-prec-sqrt
- **-ipo**
  - inlining, loop counts, alignment information
- **-xavx**
  - to use the AVX instructions on Intel CPUs (better then –mavx)

ifort –O3 –ipo –xavx

- **-fargument-noalias**
  - assume function arguments not aliased
- **-fansi-alias**
  - assume different data types not aliased
- **-fno-alias**
  - assume pointers not aliased (dangerous!)
- **–restrict**
  - or “restrict” keyword, –std=c99

INTEL SPECIFIC COMPILER PRAGMAS

#pragma	Description
vector/novector always (un)aligned (non)temporal (no)vecremainder	Instructs the compiler to vectorize override the cost model, and vectorize non-unit strides or very unaligned memory accesses; use of streaming stores vectorize remainding loop
ivdep	The compiler is instructed to ignore not proven dependencies. However still performs a dependency analysis, and will not vectorize if it finds a proven dependency that would affect results. (e.g. for indirect addressing)
simd vectorlength(*n1*[, *n2*]...) vectorlengthfor(*data type*) (first/last)private(*var1*[, *var2*]...) reduction(*oper:var1*[,*var2*]…) linear(*var1:step1*[,*var2:step2*]...) (no)vecremainder	Compiler skips dependency analysis that might cause incorrect results after vectorization. Compilation fails if not vectorized implies the loop unroll factor from OMP parallel do syntax For every iteration var is incremented by step. Every iteration of the vectorloop var is incremented by VL*step specify different strides for different variables.

INTEL DATA ALIGNMENT

Compiler can do best optimizations if loads /stores are aligned

- 16 Bytes boundaries for SSE, 32Byte for AVX and 64Bytes for MIC
- Could be imposed for static data
- (v)movupd (vectorload) if unaligned 2x128 stores may be faster than 1x256 unaligned store.

```
__declspec(align(32)) X[1000];
void foo(float *restrict a, … )
__assume_aligned(a,32)
__assume(n1%8=0);
__assume(n2%8=0);
for(i=0;i<n;i++) X[i] += a[i+n1]
}
```

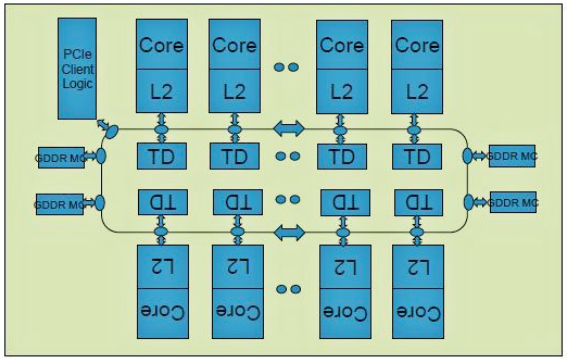
- Use special malloc libraries
- Use in C void* _mm_malloc (int size, int n)
  - Compiler creates an n-byte boundary aligned pointer to memory.
- C __declspec(align(n, [offset])) Fortran !dir$ attributes align:n::varname
  - Compiler creates the variable aligned on an “n”-byte boundary, with an “offset” in bytes.

- C __assume_aligned(a,n) Fortran !dir$ assume_aligned varname:n
  - Instructs the compiler to assume that array a is aligned on an n
- #pragma vector aligned
  - Vectorize using aligned loads /stores for vector accesses

XEON PHI : LES MODES D'UTILISATION

MODE NATIF

- Pas de modification du code
- Compilation avec l'option –mmic
- Copier le binaire sur le MIC
- SSH sur le MIC
- Exécution du binaire



MODE OFFLOAD

- Insertion de pragma dans le code
- Compilation avec l'option –offload-build
- Exécution du binaire depuis l'hôte

```
#pragma offload target(mic)
{
   sgemm(&transa, &transb, &M, &N, &K, &alpha, A,
         &LDA, B, &LDB, &beta, C, &LDC);
}
```

MODE AUTOMATIC OFFLOAD

- Pas de modification de code
- Fonctionne pour la MKL
- export MKL_MIC_ENABLE=1

FORTRAN OFFLOAD USING EXPLICIT COPIES

	Fortran Syntax	Semantics
Offload directive	`!dir$ omp offload <clause> <OpenMP construct>`	Execute next OpenMP* parallel construct on Intel® MIC Architecture
	`!dir$ offload <clauses> <statement>`	Execute next statement (function call) on Intel® MIC Architecture
Keyword for variable/function definitions	`!dir$ attributes offload:<MIC> :: <rtn-name>`	Compile function or variable for CPU and Intel® MIC Architecture
Data transfer	`!dir$ offload_transfer target(mic)`	Initiates asynchronous data transfer, or initiates and completes synchronous data transfer

GNU COMPILERS OPTIMIZATIONS

Very popular compiler and tools
GNU GCC 4.7 March 2012 latest version : GCC 4.8

GNU GCC compiler suite

- Supports a huge number of front-ends/back-end combinations …
- Fortran compiler has missing features (gfortran)
- C Compiler in good collaboration with Intel teams
- C++ drives and pushes the community standards (boost)

- Needs the latest mtune/march/with-cpu options available designed for AVX and Intel's newest CPUs.
  - –mtune=corei7-avx –mavx
  - Build with --with-mfpmath=avx to use AVX floating-point arithmetic
- Most optimizations are only enabled if an -O level is set on the command line.
  - optimization with very conservative defaults
  - O and –O2 will not increase the code size, and work everywhere
  - gfortran/gcc auto-vectorization needs –O3
  - Profile feedback available

GNU COMPILERS LOOP OPTIMIZATIONS

To use this code transformation, GCC has to be configured with --with-ppl and --with-cloog to enable the Graphite loop transformation infrastructure.

-floop-interchange

Perform loop interchange transformations on loops.

-floop-strip-mine (use with loop-block-tile-size parameter for striplength)

Loop blocking of a single loop. Strip mining splits a loop into two nested loops. The outer loop has strides equal to the strip size and the inner loop has strides of the original loop within a strip.

-floop-block

Perform loop blocking transformations on loops. Blocking strip mines each loop in the loop nest such that the memory accesses of the element loops fit inside caches.

```
DO II = 1, N, 51
 DO JJ = 1, M, 51
  DO I = II, min (II + 50, N)
   DO J = JJ, min (JJ + 50, M)
      A(J, I) = B(I) + C(J)
   ENDDO
  ENDDO
 ENDDO
ENDDO
```

```
DO II = 1, N, 51
    DO I = II, min (II + 50, N)
          A(I) = A(I) + C
    ENDDO
ENDDO
```

GNU COMPILERS AGGRESSIVE OPTIONS

-ffast-math

- Allows mathematical simplifications for computations.
- May be needed for vectorization.

-ftree-loop-distribution, -ftree-vectorize

- Perform loop distribution : one loop is distributed into several smaller loops.
- allow further loop optimizations, like parallelization or vectorization, to take place.

-ftree-vectorizer-verbose=2

- vectorization report

C may need the restrict qualifier for the pointers and intrinsic __builtin_assume_aligned

```
void test4(double * restrict a, double * restrict b)
{
int i;
double *x = __builtin_assume_aligned(a, 16);
double *y = __builtin_assume_aligned(b, 16);
for (i = 0; i < SIZE; i++) { x[i] += y[i]; }
}
```

PGI COMPILERS

PGI Fortran pgf77, pgf90, pgf95
PGI C/C++ pgcc pgCC

-fast includes "-O2 –Munroll –Mnoframe –Mlre"
-fastsse includes "-fast –Mvec=simd –Mcache_align"
-Mipa=fast,inline
enables inter-procedural analysis optimization and function inlining
-Mpft … -Mpfo
enables profile and data feedback based optimizations
-Minline inline functions and subroutines
-tp bulldozer-64 sandybridge-64 x64
target processor architecture : x64 for AMD and Intel x64 processors

PGI Cuda Fortran (with offload directives)
PGI accelerator : –tp target-host-processor –ta=nvidia
enables accelerator directives and target the NVIDIA GPUs
(cuda 2.3 to cuda 5.0 and fermi, tesla, kepler)
–acc enables the openacc directives and the openacc runtime.

PGI COMPILERS REPORTS

–Minfo

- accel
  - accelerator information.
- ftn, inline, lre, loop (with vectorization), ipa ,mp, opt, par, time, unroll, vect
- lre
  - loop-carried redundancy elimination
- intensity
  - messages about the intensity of the innermost loop :
  - For floating point loops:
    number of floating point operations / (number of floating point loads + stores)
  - For integer loops :
    total number of integer arithmetic operations / (total number of integer loads + stores)

Runtime information of GPU runs (Linux time command)

-ta=nvidia,time

collects and prints out simple timing information about the accelerator regions and generated kernels

```
Accelerator Kernel Timing Data
15: region entered 1 times
time(us): total=1490738 …
```

PGI CUDA FORTRAN CALL KERNEL

```
CUDA Fortran VADD Host Code
subroutine vadd( A, B, C )
use kmod
real(4), dimension(:) :: A, B, C
real(4), device, allocatable,
dimension(:):: &
Ad, Bd, Cd
integer :: N
N = size( A, 1 )
allocate( Ad(N), Bd(N), Cd(N) )
Ad = A(1:N)
Bd = B(1:N)
call vaddkernel<<<(N+31)/32,32>>>
( Ad, Bd, Cd, N )
C(1:N) = Cd
deallocate( Ad, Bd, Cd )
end subroutine
```

```
CUDA Fortran VADD Device Code
module kmod
use cudafor
Contains

attributes(global) subroutine
vaddkernel(A,B,C,N)
real(4), device :: A(N), B(N), C(N)
integer, value :: N
integer :: i
i = (blockidx%x-1)*blockdim%x + threadidx
%x
if( i <= N ) C(i) = A(i) + B(i)
end subroutine
end module
```

The launch is asynchronous host program continues, may issue other launches

PGI CUF KERNELS

```
subroutine vadd( A, B, C )
use kmod
real(4), dimension(:) :: A, B, C
real(4), device, allocatable,
dimension(:):: &
Ad, Bd, Cd
integer :: N
N = size( A, 1 )
allocate( Ad(N), Bd(N), Cd(N) )
Ad = A(1:N)
Bd = B(1:N)
!$cuf kernel do(1) <<< (N+31)/32, 32>>>
do i = 1, n
Cd(i) = Ad(i) + Bd(i)
enddo
C(1:N) = Cd
deallocate( Ad, Bd, Cd )
end subroutine
```

CUF Kernels

Enabled with pragma

!$cuf kernel do (loop depth)

do by itself defaults to do(1)

Chevron syntax <<< grid, block>>>

one entry for each loop,

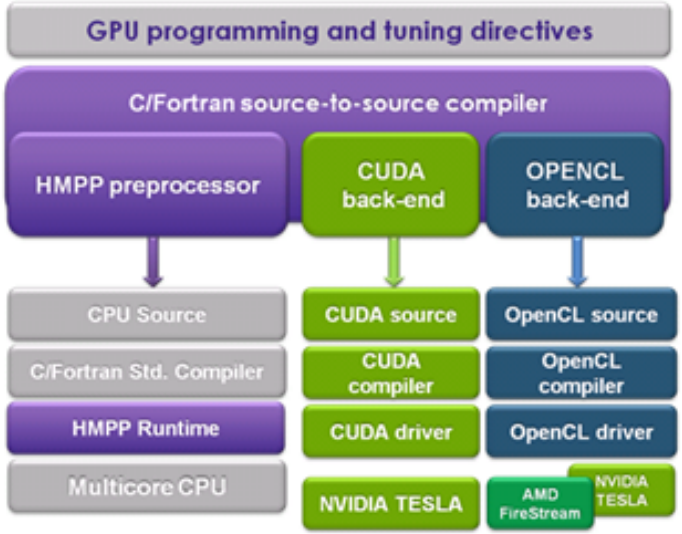
innermost first

blocks or threads may be *

body of the loop(s) become the body of the kernel

enable –Minfo to see compiler messages

CAPS HMPP




```
!$acc region
  do k = 1,n1
    do i = 1,n3
      c(i,k) = 0.0
      do j = 1,n2
        c(i,k) = c(i,k) + a(i,j) * b(j,k)
      enddo
    enddo
  enddo
!$acc end region
```

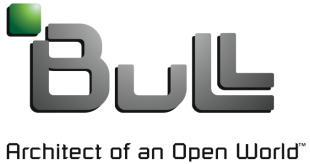
- Programmation à l'aide de pragma
- Génération automatique du code CUDA ou OpenCL
- Rapport Investissement / Performance intéressant
- Performances en deçà d'une approche directe

CONCLUSION

- Intel and AMD need vectorization
    - Compilers can vectorize code easier in Fortran
    - Compiler pragmas
- Intel proposes a straightforward programming model for the MIC
    - No easy solution for GPUs
- GNU C/C++ compilers deliver good performance if flags are set
- PGI compiler offers GPU solutions
    - Cuda Fortran
    - GPU offload directives
    - GPU profiling tool
- All compilers struggle with alignment

THANK YOU !

GUNTER.ROETH@BULL.NET



FORTRAN AND C/C++ COMPILERS

Freeware

- GNU Fortran and C/C++ compilers : gfortran, gcc, g++ are popular
  gnu.org
- Oracle Solaris Studio
  oracle.com/technetwork/server-storage/solarisstudio/downloads/index.html
- Open64
  open64.net

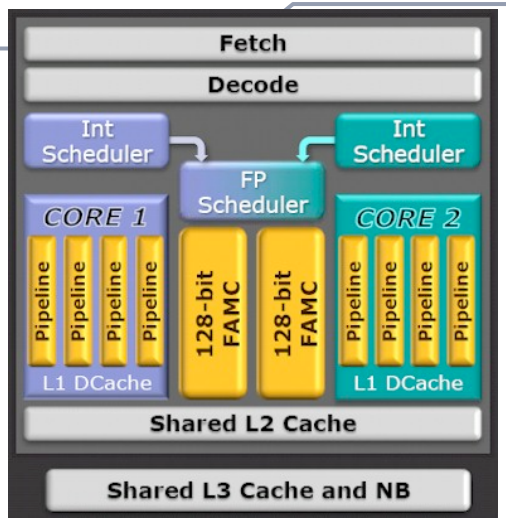
Proprietary

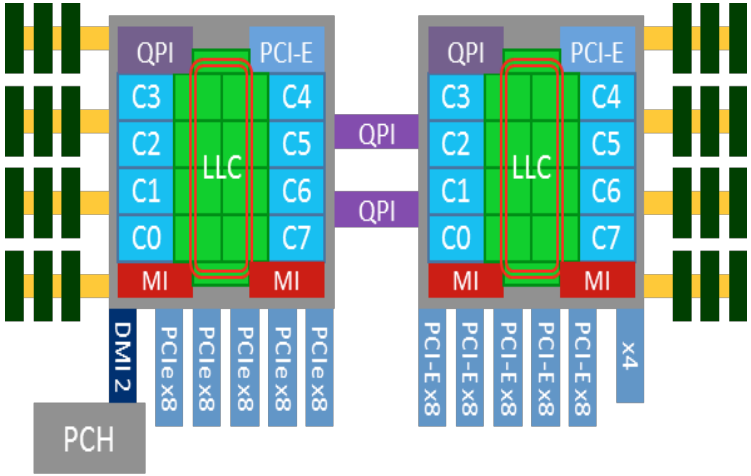
- Intel
  intel.com
- Portland Group
  pgroup.com
- Absoft, Cray, HP, IBM, NAG, PathScale, …

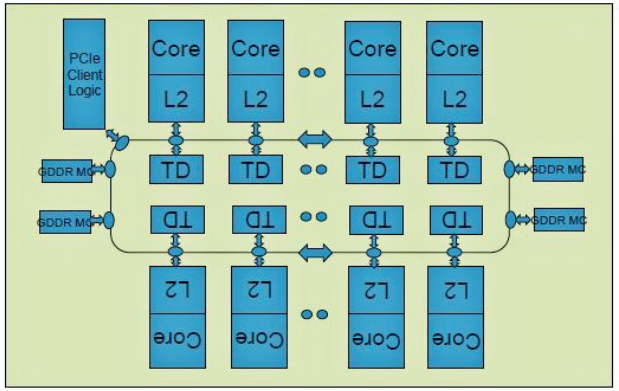
COMPILER TARGETS : INTEL AND AMD

Intel SandyBridge L5330, dual socket 8 cores per socket
Intel MIC Xeon Phi Co-processor >50 cores on a socket
AMD Bulldozer 2 integer cores sharing a FP unit 16 cores per socket

Very different CPU architecture
all processors need vectorization to run at highest possible speed !







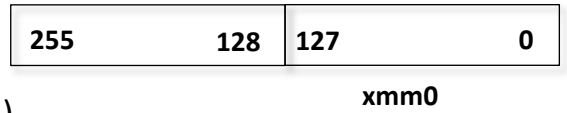
Architect of an Open World

INTEL AND AMD SSE/AVX

2007: Intel (Woodcrest) and AMD (Barcelona) SSE 128bit (data paths and FP units)
2x speed-up of vectorized codes for DP using packed SSE instructions
Chips suffering from peak transfer rate vs peak FP performance (L1,L2,Memory)

2011: Intel (SandyBridge) and AMD (Bulldozer) AVX 256bit
again a possible 2x speed-up of vectorized codes …
Impressive : with even more cores, the peak transfer rate vs peak FP performance has slightly improved (additional ports).
VEX prefix 256 SIMD support can run 3 or 4 operand syntax (compared to 2 for x86 ISA)
Less register to register copies.

The 2x can not be reached for real applications.
From Westmere (Nehalem) to Sandybridge not all components scale 2x (especially the L2/L3 cache bandwidth)
LMBench. Expect a 1.2-1.4x !



AMD shares FP unit for 2 cores. AMD extends FMA4 instructions (fused multiply-add).
AMD can run the threads using floating points in 256-bit AVX mode (scheduling ymm-based operations over the entire FP unit) or can use just one lane of the shared FP unit (xmm 128-bit) with VEX or SSE code.

INTEL LOOP OPTIMIZATIONS

Intel®Compilers: Loop Optimizations icc or ifort **–O3 (-O2)**

–Automatic vectorization (use of packed SIMD instructions)
–Loop interchange (for more efficient memory access)
–Loop unrolling (more instruction level parallelism)
–Prefetching (for patterns not recognized by h/w prefetcher)
–Cache blocking (for more reuse of data in cache)
–Loop peeling (allow for misalignment)
–Loop versioning (for loop count; data alignment; runtime dependency tests)
–Memcpy recognition (call Intel‟s fast memcpy, memset)
–Loop splitting (facilitate vectorization)
–Loop fusion (more efficient vectorization)
–Scalar replacement (reduce array accesses by scalar temps)
–Loop rerolling (enable vectorization)
–Loop reversal (handle dependencies)

-fast (same as: -ipo –O3 -no-prec-div -static -xHost)

INTEL ELEMENTAL FUNCTIONS

```
%cat example.c
#pragma simd
for(i=0; i<N; i++){
A[i] = B[i]+C[i];
}
```

Becomes Elemental Function :

```
__declspec(vector(uniform(B,C), linear(i:1)))
float foo(float *B, float *C, int i) {
return B[i]+C[i]; }
…
for(i=0; i<N; i++){
A[i] = foo(B, C, i);
}
```

- i is a sequence of integers [i, i+1, i+2…]
- B[i] nad C[i] a unit-stride load/store

```
__declspec(vector)
__declspec(vector(uniform(b,c))
float vmul(float a, float b, float c)
{ return sqrt(a)*sqrt(b) + sqrt(c); }
```

- Compiler generates different versions
- adapts depending of call ….

```
%icc -c example.c -O3 -vec_report2  -restrict -vec:simd
```

SELECTING SIMD FEATURE MAKES DIFFERENCE!

```
void foo(double* A, double * B, int n)
{
  int i;
  for (i = 0; i < n; i++) {
    A[i] += abs(B[i]);
  }
}
```

-xSSE2 (default)

```
..B2.2:
  cvttpd2dq (%rsi,%rax,8), %xmm1
  pxor      %xmm0, %xmm0
  pcmpgtd   %xmm1, %xmm0
  pxor      %xmm0, %xmm1
  psubd     %xmm0, %xmm1
  cvtdq2pd  %xmm1, %xmm2
  addpd     (%rdi,%rax,8), %xmm2
  movaps    %xmm2, (%rdi,%rax,8)
  addq      $2, %rax
  cmpq      $1000, %rax
  jb        ..B2.2
```

-xSSE4.2

```
..B2.2:
  cvttpd2dq (%rsi,%rax,8), %xmm0
  pabsd     %xmm0, %xmm1
  cvtdq2pd  %xmm1, %xmm2
  addpd     (%rdi,%rax,8), %xmm2
  movaps    %xmm2, (%rdi,%rax,8)
  addq      $2, %rax
  cmpq      $1000, %rax
  jb        ..B2.2
```

-xAVX

```
..B2.2:
  vcvttpd2dqy (%rsi,%rax,8), %xmm0
  vpabsd      %xmm0, %xmm1
  vcvtdq2pd %xmm1, %ymm2
  vaddpd      (%rdi,%rax,8), %ymm2, %ymm3
  vmovupd     %ymm3, (%rdi,%rax,8)
  addq        $4, %rax
  cmpq        $1000, %rax
  jb          ..B2.2
```

INTEL SPECIFIC COMPILER PRAGMAS

#pragma	Description
prefetch	compiler prefetches arrays of a certain type at a certain distance (on MIC only)
distribute point	Advise where to split loops: Placed before a loop, the compiler will attempt to distribute the loop based on its internal heuristic. Placed within a loop, the compiler will attempt to distribute the loop at the point of the pragma. All loop-carried dependencies will be ignored.
unroll, unroll(n), nounroll	Place before an inner loop (ignored on non-inmost loops). #pragma unroll without a count allows the compiler to determine the unroll factor. #pragma unroll(n) tell the compiler to unroll the loop n times.
loop count n loop count n1, n2, n3 … loop count min=<l>, avg=<a>, max=<u>	Hint to the compiler on expected loop iteration count: (n): always n (n1, n2, n3 …) either n1, n2, n3 … expected minimum count <l>, average count <a> and maximal count <u> Affects software pipelining, vectorization and other loop transformations.
unroll_and_jam	Hint to the compiler to partially unroll one or more loops higher in the nest than the innermost loop and fuse/jam the resulting loops back together. This pragma is not effective on innermost loops. Ensure that the immediately following loop is not the innermost loop after compiler-initiated interchanges are completed.

C OFFLOAD USING EXPLICIT COPIES

	C/C++ Syntax	Semantics
Offload pragma	`#pragma offload <clauses> <statement block>`	Allow next statement block to execute on Intel® MIC Architecture or host CPU
Keyword for variable & function definitions	`__attribute__((target(mic)))`	Compile function for, or allocate variable on, both CPU and Intel® MIC Architecture

Entire blocks of code	`#pragma offload_attribute(push, target(mic))`
	⋮
	`#pragma offload_attribute(pop)`
		Mark entire files or large blocks of code for generation on both host CPU and Intel® MIC Architecture

Data transfer	`#pragma offload_transfer target(mic)`	Initiates asynchronous data transfer, or initiates and completes synchronous data transfer

OFFLOAD USING EXPLICIT COPIES – CLAUSES

Variables and pointers restricted to scalars, structs of scalars, and arrays of scalars

Clauses	Syntax	Semantics
Target specification	`target( name[:card_number] )`	Where to run construct
Conditional offload	`if (condition)`	Boolean expression
Inputs	`in(var-list modifiers`$_{opt}$`)`	Copy from host to coprocessor
Outputs	`out(var-list modifiers`$_{opt}$`)`	Copy from coprocessor to host
Inputs & outputs	`inout(var-list modifiers`$_{opt}$`)`	Copy host to coprocessor and back when offload completes
Non-copied data	`nocopy(var-list modifiers`$_{opt}$`)`	Data is local to target
Async. offload	`signal(signal-slot)`	Trigger async offload
Async. offload	`wait(signal-slot)`	Wait for completion

OFFLOAD USING EXPLICIT COPIES – MODIFIERS

Variables and pointers restricted to scalars, structs of scalars, and arrays of scalars

Modifiers	Syntax	Semantics
Specify pointer length	`length(element-count-expr)`	Copy N elements of the pointer's type
Control pointer memory allocation	`alloc_if ( condition )`	Allocate memory to hold data referenced by pointer if condition is TRUE
Control freeing of pointer memory	`free_if ( condition )`	Free memory used by pointer if condition is TRUE
Control target data alignment	`align ( expression )`	Specify minimum memory alignment on target

Simple Example of Extending Programming to Support MIC

```
# DEFINE NSET 1000000
INT MAIN ( INT ARGC, CONST CHAR** ARGV )
{     LONG INT I;
      FLOAT NUM_INSIDE, PI;
      NUM_INSIDE = 0.0F;
      #PRAGMA OFFLOAD TARGET (MIC)
   #PRAGMA OMP PARALLEL FOR REDUCTION(+:NUM_INSIDE)
      FOR( I = 0; I < NSET; I++ )
      {     FLOAT X, Y, DISTANCE_FROM_ZERO;
                  // GENERATE X, Y RANDOM NUMBERS IN [0,1)
                  X = FLOAT(RAND()) / FLOAT(RAND_MAX + 1);
                  Y = FLOAT(RAND()) / FLOAT(RAND_MAX + 1);
                  DISTANCE_FROM_ZERO = SQRT(X*X + Y*Y);
                  IF ( DISTANCE_FROM_ZERO <= 1.0F )
                  NUM_INSIDE += 1.0F;
            }
   PI = 4.0F * ( NUM_INSIDE / NSET );
   PRINTF("VALUE OF PI = %F \N",PI);
}
```

A one line change from the CPU version

PGI COMPILERS PRAGMAS

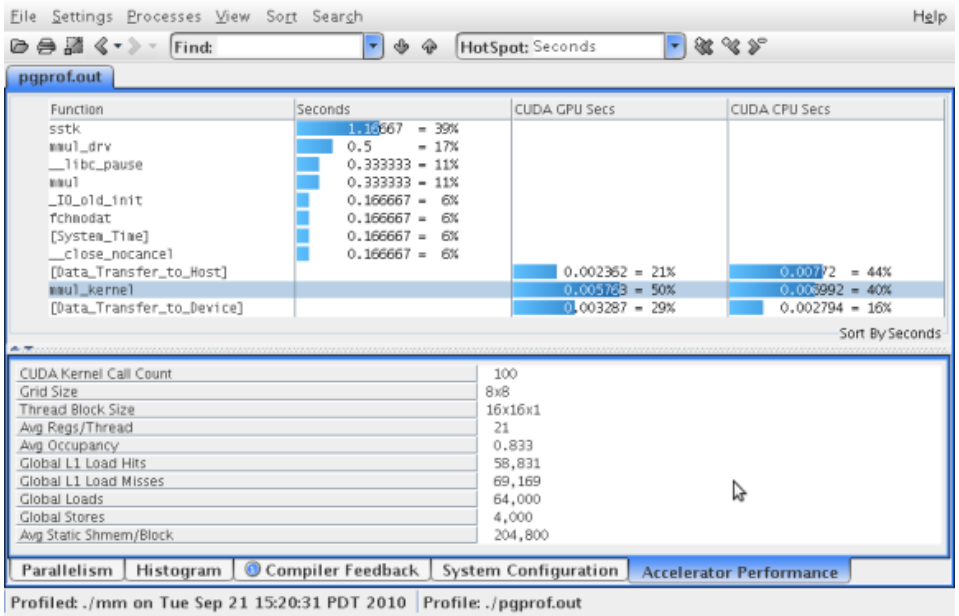
PGI has a very long list of possible pragmas (guiding optimisation, code generation, parallelisation)
The general syntax of a pragma is:
C #pragma [scope] pragma-body
Fortran pgi$g directive !pgi$r directive !pgi$l directive !pgi$ directive

The valid scopes are: Global, Routine, Loop

- altcode
  - generate alternate code for vectorized and parallelized loops.
- ivdep
  - ignore potential data dependencies.
- prefetch
  - control how prefetch instructions are emitted
- safe
  - treat pointer arguments as safe.
- safeptr
  - ignore potential data dependencies to pointers.
- vector
  - vectorizations

CUDA FORTRAN PROFILER

GUI …. or … runtime info



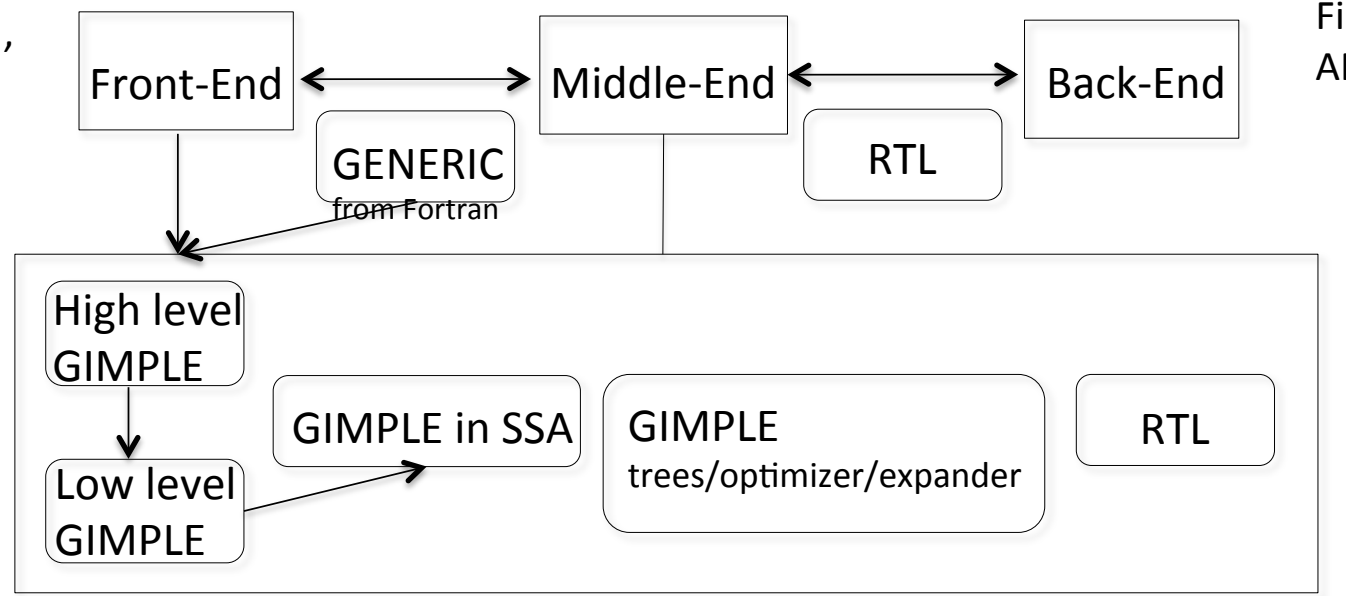
Profiling GPU runs
-ta=nvidia,time
collects and prints out simple timing information about the
accelerator regions and generated kernels :

```
Accelerator Kernel Timing Data
bb04.f90
s1
15: region entered 1 times
time(us): total=1490738
init=1489138 region=1600
kernels=155 data=1445
w/o init: total=1600 max=1600
min=1600 avg=1600
18: kernel launched 1 times
time(us): total=155 max=155 min=155
avg=155
```

COMPILERS (GCC)

- Front-end is the language interface
  - No optimisation other than the user to write good code
- Middle and Back-End are language-independent.
- Code optimisations are done in the middle-end with IR
- Register allocation, linearization and assembly code generation in the back-end

C, C++, Fortran,
Ada, Go



Assembler
File X86-64, Sparc
ARM

GNU COMPILERS RESTRICT AND ALIGNMENT OPTIONS

C may need the restrict qualifier for the pointers and intrinsic __builtin_assume_aligned

```
void test4(double * restrict a, double * restrict b)
{
int i;
double *x = __builtin_assume_aligned(a, 16);
double *y = __builtin_assume_aligned(b, 16);
for (i = 0; i < SIZE; i++) { x[i] += y[i]; }
}
```