

IBM Software Group

IBM's BlueGene/L Compiler Implementation

Mark Mendell
Roch Archambault

IBM Toronto Lab

October 14, 2003 | Blue Gene/L Workshop, Reno NV



Outline

- IBM Linux/PPC compilers
- Generating Code For Blue Gene/L
 - Superword Level Parallelization
 - Code Examples
- Complex Arithmetic
- SIMDization and Alignment
- Questions



IBM Linux/PPC compilers

- C, C++, Fortran
- Supports SLES 8.0, RHEL 3 (forthcoming)
- 32-bit and 64-bit Support
- Automatic Parallelization
- Symbolic Debugging support with gdb
- Exploitation of POWER4
- Portfolio of Optimizing Transformations
 - Comprehensive path length reduction
 - Whole program analysis
 - Loop optimization for parallelism, locality and instruction scheduling
 - Tuned support for all pSeries processors
- Supports –qarch=440 and –qarch=440d



IBM XL Fortran on Linux/PPC

- Based on XL Fortran for AIX, Version 8.1 Product
- Fully compliant Fortran 77/90/95 compiler
- Partial Fortran 200x Support
 - Allocatable components, IEEE module
- Many Industry Extensions
 - Cray, DEC, IBM VS
- Optimized OpenMP Fortran Version 2.0 Support



VisualAge C and C++ on Linux/PPC

- Based on VisualAge C++ for AIX, Version 6.0
- Fully compliant ISO C 1999 and ISO C++ 1998 Standards
- GNU C/C++ language and option compatibility (subset)
- Binary Compatibility with GNU C/C++ Version 3.2
 - Leverage GNU C++ Version 3.2 Runtime
- Optimized OpenMP C/C++ API Version 1.0 support



Superword Level Parallelization (SLP)

- S. Larsen and S. Amarasinghe. *Exploiting Superword Level Parallelism with Multimedia Instruction Sets*. In Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation, Vancouver, B.C., June 2000
- Designed for MMX and AltiVec type instruction sets (symmetrical)
- Basic SLP Algorithm:
 - Find aligned loads/stores in a basic block and pair them up
 - Follow use/def chains to find pairs of instructions that can be parallelized, using a estimated benefit function
 - Combine pairs of instructions to form larger groups
 - Schedule the instructions in a basic block



SLP for BlueGene/L

- No need to combine pairs of instructions, since M in SIMD is 2
- Many asymmetrical instructions that greatly complicate evaluation of the benefit of each instruction.
- Each symbolic register is marked as primary, secondary, or unknown
- Benefit function checks to see if a register is in the correct location for each instruction. Unknown operands are assumed to be correct (register allocator will fix up). Cost of moving between primary and secondary registers is 5 cycles.
- Load/store parallel/cross must not cross cache line boundary
 - Must ‘prove’ alignment is safe to generate quad load/stores



Code Example

```
_Complex double a[SIZE], b[SIZE];
void p ()
{
    int i;
    for (i = 0; i < SIZE; i++)
        a[i] += b[i];
}
```



Before Blue Gene/L Code Generation

6: CL.3:

```
7: LFL    *fp407=a[]0#re(gr423,8)
7: LFL    *fp408=a[]0#im(gr423,16)
7: LFL    *fp410=b[]0#re(gr425,8)
7: LFDU   *fp411,*gr425=b[]0#im(gr425,16)
7: AFL    *fp413=fp407,fp410,fcr
7: AFL    *fp414=fp408,fp411,fcr
7: STFL   a[]0#re(gr423,8)=fp413
7: STFDU  *gr423,a[]0#im(gr423,16)=fp414
6: BCT    ctr=CL.3,taken=80%(80,20)
```

Optimizations disabled: unrolling, modulo scheduling, global scheduling



SLP: Find Pairs of Aligned Instructions

6: CL.3:

7: LFL *fp407=a[]0#re(gr423,8)
7: LFL *fp408=a[]0#im(gr423,16)
7: LFL *fp410=b[]0#re(gr425,8)
7: LFDU *fp411,*gr425=b[]0#im(gr425,16)
7: AFL *fp413=fp407,fp410,fcr
7: AFL *fp414=fp408,fp411,fcr
7: STFL a[]0#re(gr423,8)=fp413
7: STFDU *gr423,a[]0#im(gr423,16)=fp414
6: BCT ctr=CL.3,taken=80%(80,20)



SLP: Use Use/Def to Find Parallel Operations

6: CL.3:

7: LFL *fp407=a[]0#re(gr423,8)
7: LFL *fp408=a[]0#im(gr423,16)
7: LFL *fp410=b[]0#re(gr425,8)
7: LFDU *fp411,*gr425=b[]0#im(gr425,16)
7: AFL *fp413=fp407,fp410,fcr
7: AFL *fp414=fp408,fp411,fcr
7: STFL a[]0#re(gr423,8)=fp413
7: STFDU *gr423,a[]0#im(gr423,16)=fp414
6: BCT ctr=CL.3,taken=80%(80,20)



SLP: Output from BG/L Code Generation

```
6: CL.3:  
7: LFPL  *fp407,*fp408=a[]0#re(gr423,8)  
7: LFPL  *fp410,*fp411=b[]0#re(gr425,8)  
7: AI    *gr425=gr425,16  
7: FPADD *fp413,*fp414=fp407,fp408,fp410,fp411,fcr  
7: SFPL  a[]0#re(gr423,8)=fp413,fp414  
7: AI    *gr423=gr423,16  
6: BCT   ctr=CL.3,taken=80%(80,20)
```



Final Generated Code

```
7: LI      gr5=8
6: CL.3:
7: LFPL   fp1(fp33=a[0#re(gr3,gr5,0)
7: LFPL   fp0(fp32=b[0#re(gr4,gr5,0)
7: AI      gr4=gr4,16
7: FPADD fp0(fp32=fp1(fp33,fp0(fp32,fcr
7: SFPL   a[0#re(gr3,gr5,0)=fp0(fp32
7: A       gr3=gr3,16
6: BCT    ctr=CL.3,taken=80%(80,20)
```



Final Generated Code (With all optimizations)

7: CL.27:

```
7:   FPADD    fp5,fp37=fp0,fp32,fp3,fp35,fcr
7:   LFPL     fp3,fp35=a[]0#re(gr6,gr10,0,trap=72)
7:   LFPL     fp0,fp32=b[]0#re(gr5,gr3,0,trap=8)
7:   AI       gr6=gr6,64
7:   SFPL     a[]0#re(gr6,gr11,0,trap=-40)=fp2,fp34
7:   FPADD    fp4,fp36=fp1,fp33,fp4,fp36,fcr
7:   LFPL     fp2,fp34=a[]0#re(gr6,gr4,0,trap=24)
7:   LFPL     fp1,fp33=b[]0#re(gr5,gr4,0,trap=24)
7:   SFPL     a[]0#re(gr6,gr7,0,trap=-24)=fp5,fp37
7:   AI       gr5=gr5,64
7:   FPADD    fp5,fp37=fp3,fp35,fp0,fp32,fcr
7:   LFPL     fp0,fp32=a[]0#re(gr6,gr12,0,trap=40)
7:   LFPL     fp3,fp35=b[]0#re(gr5,gr7,0,trap=-24)
7:   SFPL     a[]0#re(gr6,gr9,0,trap=-8)=fp4,fp36
7:   FPADD    fp2,fp34=fp2,fp34,fp1,fp33,fcr
7:   LFPL     fp1,fp33=a[]0#re(gr6,gr8,0,trap=56)
7:   LFPL     fp4,fp36=b[]0#re(gr5,gr9,0,trap=-8)
7:   SFPL     a[]0#re(gr6,gr3,0,trap=8)=fp5,fp37
0:   BCT      ctr=CL.27,taken=****
```

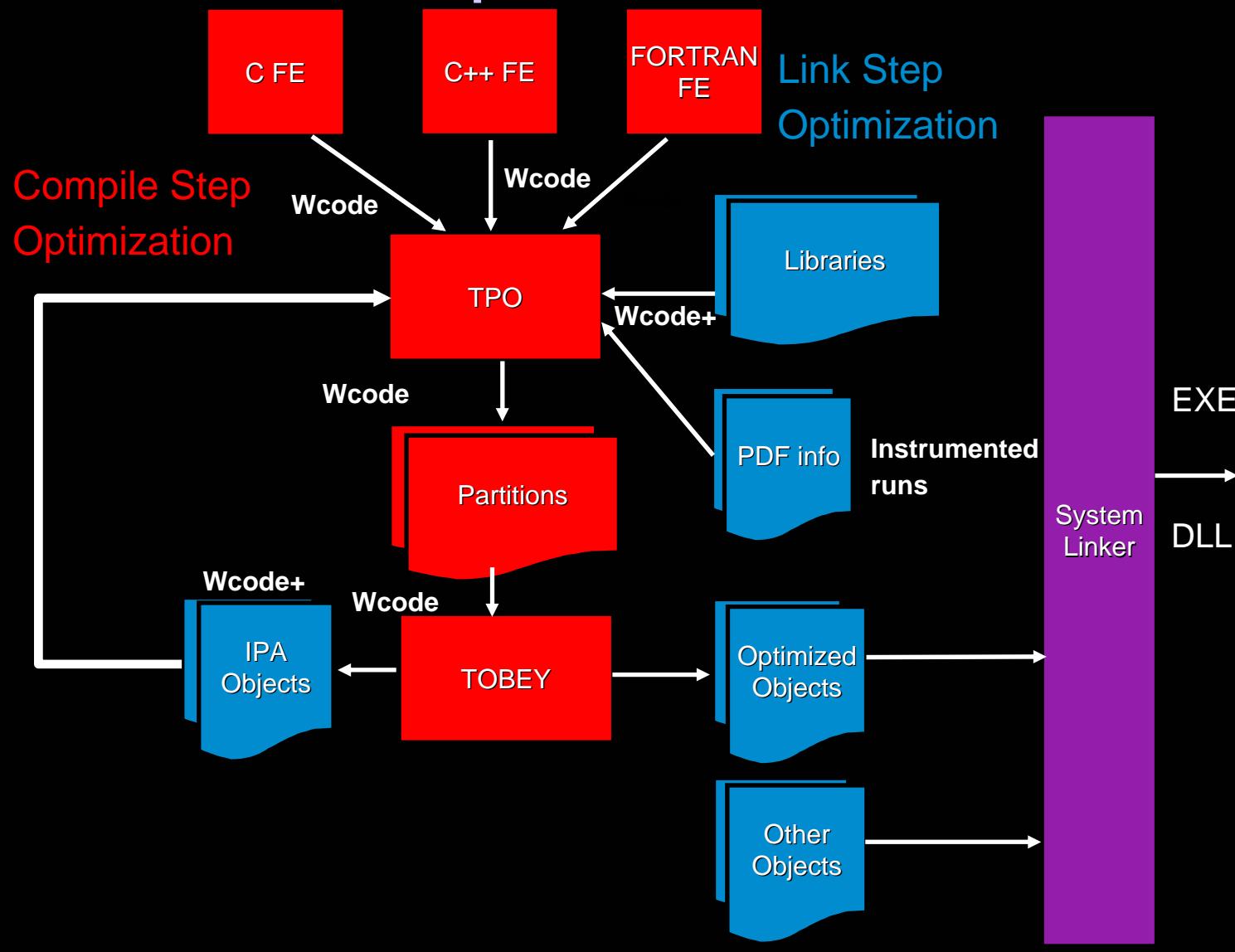


Generating code for Blue Gene/L

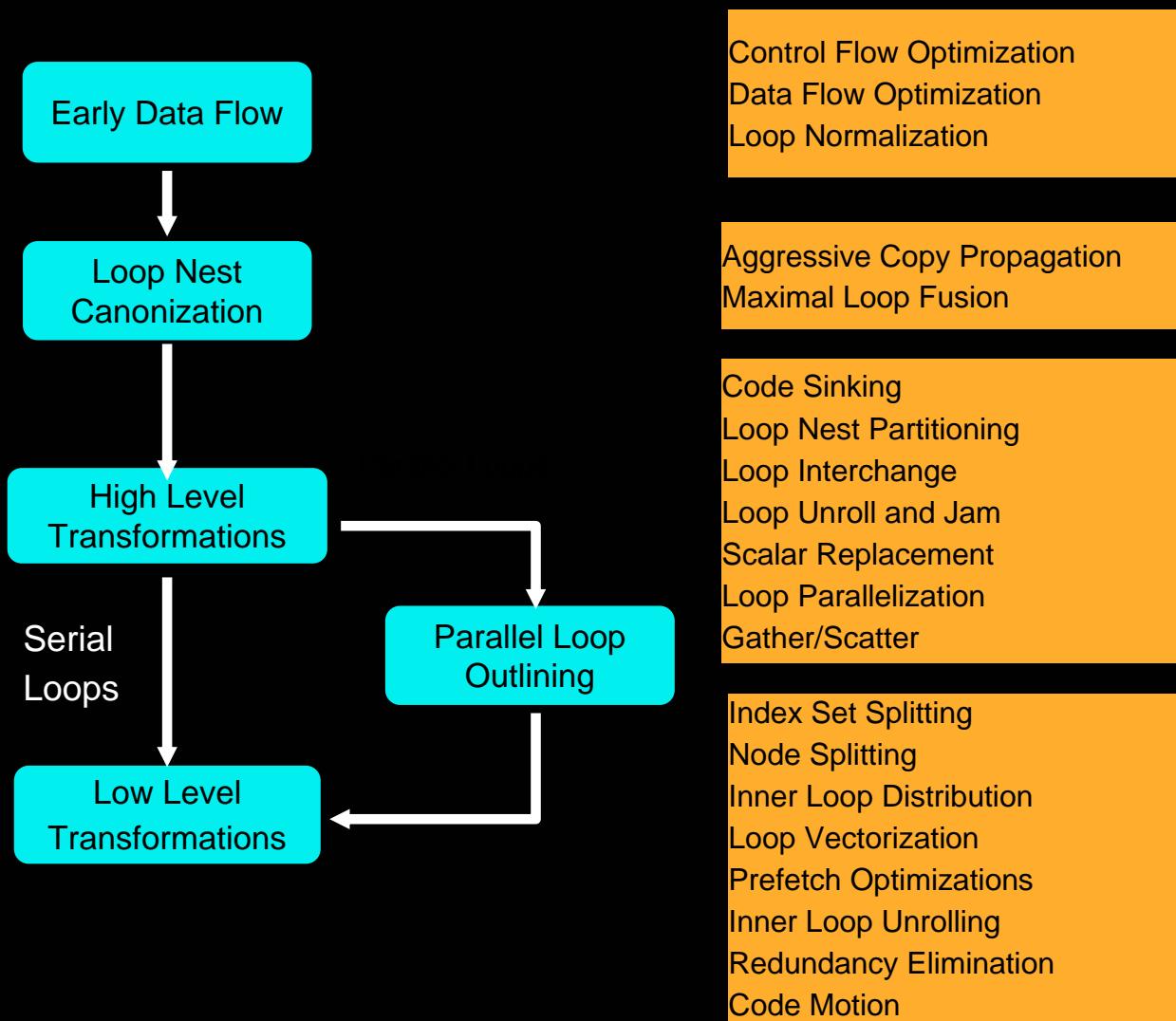
- Enhance SLP to generate asymmetric instructions
- Generate complex arithmetic using double FP instructions
- Builtins added to generate new instructions
- Scheduling information for PPC 440 core
- Anti-dependency information for register allocator
 - No register renaming
- Use quad load/store to move memory quickly
- Goal: generate good code for ESSL matmul, matvect, dotprod
- -qarch=440 or -qarch=440d / -qtue=440



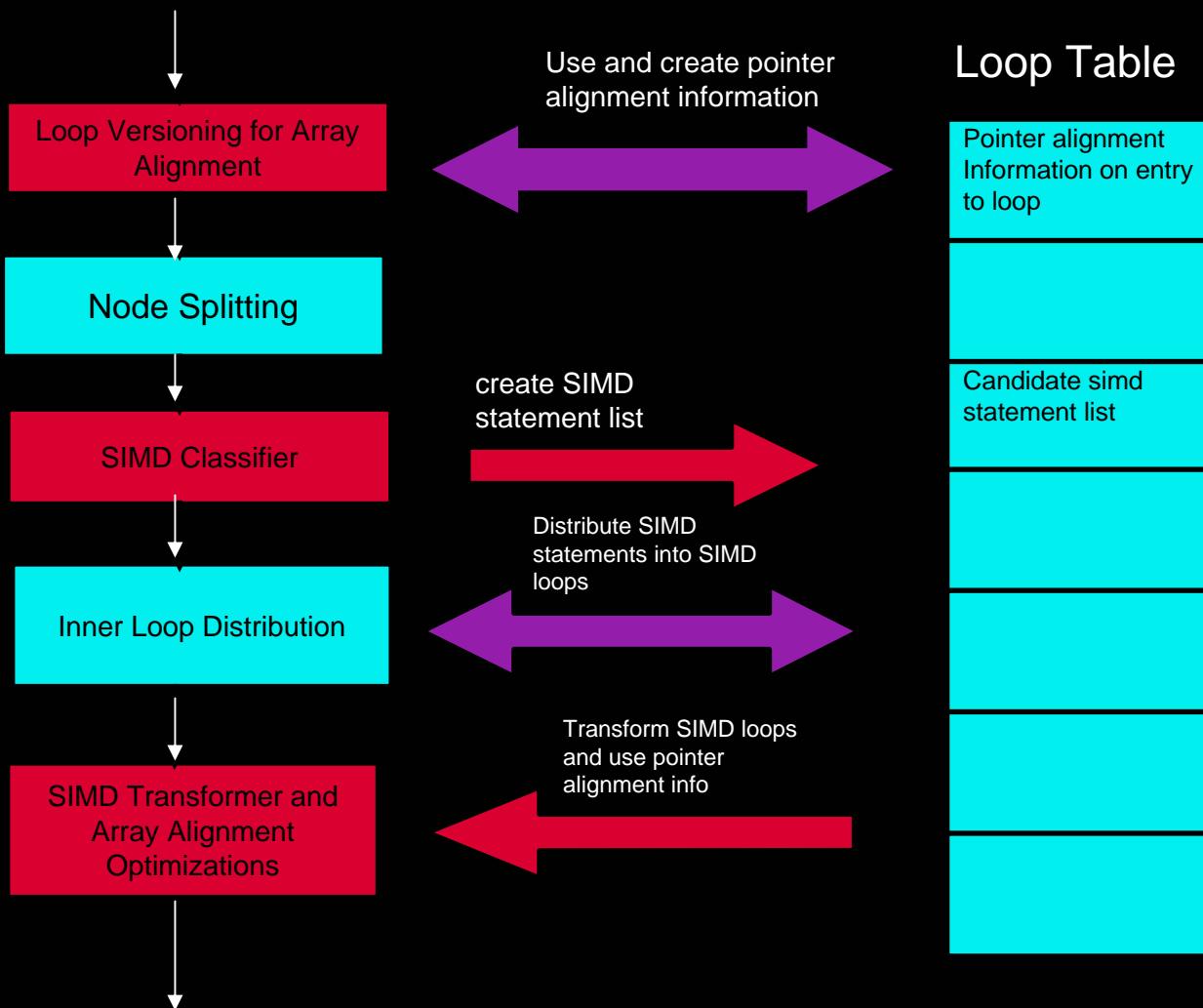
IBM Compiler Architecture



Loop Optimization Overview In TPO



Automatic SIMDization As A Low Level Transformation



Phases Of Automatic SIMDization

- Create multiple versions of inner loops:
 - One optimistic version for the case when all arrays are aligned on a 16 byte boundary
 - Use information from interprocedural pointer analysis to determine which array access is definitely aligned, definitely not aligned or maybe aligned. This can be used to determine the feasibility of versioning.
- Node splitting can be used to extract SIMDizable part of a statement. Scalar expansion is also done in this phase to enable more SIMDization.
- The SIMD classifier identifies statements which can be fully transformed to SIMD code. Each loop table entry is annotated with the list of candidate SIMDizable statements.



Example Source Code

```
void zaxpy (double _Complex x[], double _Complex y[],  
            double _Complex a, int n)  
{  
    #pragma disjoint(*x,*y)  
    int i;  
    for (i = 0; i < n; i++)  
        y[i] = a*x[i]+y[i];  
}
```



Example Code (Before Loop Versioning)

```
void zaxpy ( char * x, char * y, complex double a, long n ) {  
    if (!(0 < n)) goto lab_4;  
    @CIV0 = 0;  
    do {  
        y->y[]0.rns0.[@CIV0] = a * x->x[]0.rns1.[@CIV0] +  
                                    y->y[]0.rns0.[@CIV0];  
        @CIV0 = @CIV0 + 1;  
    } while ((unsigned) @CIV0 < (unsigned) n);  
lab_4:  
}
```



Example Code (After Loop Versioning)

```
void zaxpy ( char * x, char * y, complex double a, long n ) {
    if (!(! ((long) y & 15) & ! ((long) x & 15))) goto lab_15;
    if (!(0 < n)) goto lab_4;
    __alignx(16,y); __alignx(16,x);
    @CIV0 = 0;
    do {      y->y[]0.rns0.[@CIV0] = a * x->x[]0.rns1.[@CIV0] +
                y->y[]0.rns0.[@CIV0];
                @CIV0 = @CIV0 + 1;
    } while ((unsigned) @CIV0 < (unsigned) n);
lab_4: goto lab_16;
lab_15 : if (!(0 < n)) goto lab_18;
    @CIV0 = 0;
    do {      y->y[]0.rns0.[@CIV0] = a * x->x[]0.rns1.[@CIV0] +
                y->y[]0.rns0.[@CIV0];
                @CIV0 = @CIV0 + 1;
    } while ((unsigned) @CIV0 < (unsigned) n);
lab_18:lab_16:
```



Phases Of Automatic SIMDization (Continued)

- Inner loop distribution uses the SIMDizable statement list and other heuristics to distribute an inner loop into SIMDizable loops and non-SIMDizable loops. During distribution, statements may be reordered to maximize the size of the SIMD loops.
- The SIMD transformer will map the code from the SIMD loops into a sequence of SIMD code. Array alignment information will be propagated to this point in order to perform array alignment transformations. Here is an example of dynamic loop peeling :

Maybe misaligned

```
for (i=x; i<y; i++) {  
    a[i]  
}
```



```
for (i=x; i<y; i++) {  
    if (&a[i] % 4 == 0) break;  
    a[i]  
}  
for (; i<y; i+=4) {  
    a[i+0...3]  
}
```



Questions?

Answers?



Appendices



C/C++ Features added to support Linux

- Local Labels
- Labels as Values
- Function Attributes
- support: section, constructor, destructor, alias, weak, noreturn, const, pure
- accept and ignore: format, format_arg, no_instrument_function, no_check_memory_usage, regparm, stdcall, cdecl, longcall, dllimport, dllexport, exception, function_vector, interrupt_handler, eightbit_data, tiny_data, interrupt, model
- Type Attributes
- accept and ignore: aligned, packed, transparent_union, unused
- Variable Attributes
- support: aligned, mode, weak, packed
- accept and ignore: nocommon, section, transparent_union, unused, model
- C++ style comments



Even more C/C++ compatibility features

- Dollar signs
- Statement Expressions
- Typeof
- long long
- __alignof__
- __extension__ (accept and ignore)
- Inline
- Explicit Register Variables (accept and ignore)
- Alternate Keywords
- Generalized Lvalues (C)
- extern template
- #pragma gcc header
- #assert, #unassert, #cpu, #machine, #system
- #warning
- #include_next

