OpenMP and Vectorization in GCC

Diego Novillo
dnovillo@redhat.com
Red Hat Canada

2nd Workshop on Challenges for Parallel Computing
Toronto, Canada, October 2006
Introduction

- Overview of GCC
- OpenMP
- Automatic parallelization
- Vectorization
- Status and Future Work
GCC Overview

Front End

C
C++
Java
Fortran

Middle End

GENERIC
GIMPLE
Inter Procedural Optimizer
SSA Organizer

Back End

RTL
RTL Optimizer
Final Code Generation

Assembly

Call Graph Manager
Pass Manager
OpenMP

```c
#include <omp.h>

main()
{
    #pragma omp parallel
    printf ("[%d] Hello\n", omp_get_thread_num());
}
```

$ gcc -fopenmp -o hello hello.c
$ ./hello
[2] Hello
[3] Hello
[0] Hello

← Master thread

$ gcc -o hello hello.c
$ ./hello
[0] Hello
OpenMP Implementation

- GNU OpenMP (GOMP)
- Four components
  - Parsing
  - Intermediate Representation
  - Code Generation
  - Run time library (libgomp)
OpenMP Implementation

- C
- C++
- Fortran

New OMP nodes
Common target for FEs

Outlining
Calls to libgomp

Data mappings
Some expansion

SESE parallel regions

Expand
main()
{
    int i, sum = 0;

    #pragma omp parallel for
    for (i = 0; i < 10; i++)
    {
        #pragma omp atomic
        sum += i;
    }

    printf("sum = %d\n", sum);
}
main()
{
    int i, sum = 0;

    #pragma omp parallel shared(sum)
    {
        #pragma omp for nowait private(i)
        for (i = 0; i <= 9; i = i + 1)
            __sync_fetch_and_add_4 (&sum, i);
    }
    printf("sum = %d\n", sum);
}
main()
{
    int i, *D.1576, sum = 0;
    struct .omp_data_s .omp_data_o;

    .omp_data_o.sum = &sum;
    #pragma omp parallel shared(sum)
    .omp_data_i = &.omp_data_o;
    #pragma omp for nowait private(i)
    for (i = 0; i <= 9; i = i + 1)
        D.1576 = .omp_data_i->sum;
    __sync_fetch_and_add_4 (D.1576, i.0);
    OMP_RETURN
    OMP_RETURN
    printf (&"sum = %d\n"[0], sum.1);
}
Final expansion (main)

main()
{
    int i, sum = 0;
    struct __omp_data_s __omp_data_o;

    __omp_data_o.sum = &sum;
    __builtin_GOMP_parallel_start (main.omp_fn.0,
                                   &omp_data_o, 0);
    main.omp_fn.0 (&omp_data_o);
    __builtin_GOMP_parallel_end ();
    printf ("sum = %d\n", sum);
}
Final Expansion
(main.omp_fn.0)

main.omp_fn.0 (.omp_data_i)
{
    D.1581 = __builtin_omp_get_num_threads();
    D.1582 = (unsigned int) D.1581;
    D.1583 = __builtin_omp_get_thread_num();
    D.1584 = (unsigned int) D.1583;
    D.1585 = 10 / D.1582;
    D.1586 = D.1585 * D.1582;
    D.1587 = D.1586 != 10;
    D.1588 = D.1585 + D.1587;
    D.1589 = D.1588 * D.1584;
    D.1590 = D.1589 + D.1588;
    D.1591 = MIN_EXPR <D.1590, 10>;
    if (D.1589 >= D.1591) goto <L2> else goto <L0>

    <L0>:
    D.1592 = (int) D.1589;
    D.1593 = D.1592 * 1;
    i = D.1593 + 0;
    D.1594 = (int) D.1591;
    D.1595 = D.1594 * 1;
    D.1596 = D.1595 + 0;

    <L1>:
    D.1576 = .omp_data_i->sum;
    __sync_fetch_and_add_4 (D.1576, i);
    i = i + 1;
    D.1597 = i < D.1596;
    if (D.1597) goto <L1>; else goto <L2>;

    <L2>:
    return;
}

Iteration space partitioning

Local min/max limits
Runtime library

- Wrapper around POSIX threads
  - Various system-specific performance tweaks

- Synchronization usually 1-1 mapping except
  - `omp master` → Blocks threads with ID != 0
  - `omp single` → `copyprivate` needs special expansion to handle broadcast.

- All scheduling variants of `omp for` implemented
  - Static schedules are open coded by compiler
Auto Parallelization

• OMP codes can be emitted internally as result of analysis
  - OMP_SECTIONS → task parallelism
  - OMP_FOR → loop parallelism
  - OMP sharing clauses → data sharing
  - Synchronization with appropriate directives
• Work in progress scheduled for GCC 4.3.
Vectorization

- Traditional pattern-based implementation
  - New patterns added with every release
- Multi-platform: x86, ppc64, ia64
- Two distinct phases
  - Analysis → high-level (GIMPLE)
  - Transformation → low-level (RTL)
Vectorization

- GIMPLE extended with vector abstractions
  - Concise expression of high-level idioms
  - Reductions, saturated ops, dot products, extractions, type conversions, etc.
- RTL and target API conveys
  - Available operations
  - Costs
- Designed to simplify portability to many platforms
Vectorization

```c
int a[256], b[256], c[256];

foo ()
{
    for (i = 0; i < 256; i++)
        a[i] = b[i] + c[i];
}
```

```
.L2:
movdqa c(%eax), %xmm0
paddwb b(%eax), %xmm0
movdqa %xmm0, a(%eax)
add $16, %eax
cmpl $1024, %eax
jne .L2

.L2:
movl c(,%edx,4), %eax
addl b(,%edx,4), %eax
movl %eax, a(,%edx,4)
addl $1, %edx
cmpl $256, %edx
jne .L2
```

$ gcc -O2 -ftree-vectorize

$ gcc -O2

(~2x on P4)
Status and Future Work

- Vectorizer exists as of version 4.0
  - New patterns added with every release
  - Straight line code vectorization in progress
- OpenMP to be released with GCC 4.2
  - Full 2.5 spec implemented
  - Performance comparable to ICC (SPECOMP2001)
  - Automatic parallelization in progress
- Implementation available in Fedora Core 5