Vectorization
How to Utilize the Vector Units in a Processor
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Vector (SIMD) Hardware

• Observations:
  • Many computationally intensive application loop over data doing identical operations
    • E.g., for (i = 0; i < N; i++) A[i] += x * B[i];
  • The consecutive instructions are identical, except for values
    • Not counting the incrementing of i
  • By duplicating floating point hardware, we can make this go much faster
    • (Instruction fetch and decode are identical ... load and store are to consecutive locations)
    • Floating point units are relatively cheap ... less than 1 square mm on the chip

• Add vector register, vector FPUs, and vector instructions
  • Most modern processors provide such vector hardware
  • Made accessible through specialized instructions
  • Intel: SSE, AVX, PowerPC: AltiVec
Vectorizing Your Code

• I.e., making sure your code uses the vector units
• Ideally, compiler should do it
• Sometimes, it doesn’t
  • Because it cannot guarantee that instructions are independent, for example
• How to find out a code you expected to vectorize was not?
• What to do in that case?
  • Unblock the compiler via annotations and declarations
    • Restrict
  • Use vector “intrinsics” in your code, telling the compiler to produce vector instructions
    • SSE
Many compilers support options that report
  • Whether individual loops were vectorized
  • If not, why not?

For example, Intel compilers support a “-vec-report” command line option

A lot of information is available online

For example, see:
Annotations and Hints to the Compiler

void add(float* a, float* b, float* c) {
    for (int i = 0; i < SIZE; i++) {
        c[i] += a[i] + b[i];
    }
}

void add(float* restrict a, float* restrict b, float* restrict c) {
    for (int i = 0; i < SIZE; i++) {
        c[i] += a[i] + b[i];
    }
}

• This code may not be vectorized by the compiler
  • Because c may be pointing to a location inside the array a, for example
    • It may be erroneous to overwrite a before it is read
  • If you are sure that such aliasing is not happening in your program, you can
    tell the compiler so by using the keyword restrict
Performance Analysis: counters and timers
Performance Counters: PAPI

• Performance counters: processors record many events for analysis
  • E.g., number of cache misses
  • Very useful for performance analysis

• Example usages
  • Cache hit ratio to understand locality
  • Ratio of floating point instructions to understand floating point intensity

• Problem: different processors record differently

• PAPI: middleware that provides a consistent programming interface for performance counters
  • Most major processors

• All events are referenced by name and collected into EventSets for sampling

• Events can be multiplexed if counters are limited

• Statistical sampling is implemented by either software or hardware

• (Reference: http://cscads.rice.edu/workshops/july2007/autotune-slides-07/Terpstra-PAPI.pdf)
How to Use PAPI

```c
int EventSet = PAPI_NULL;
long long values[3];
/* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Create the Event Set */
    PAPI_create_eventset(&EventSet)
/* Add Total Instructions Executed to our EventSet */
    PAPI_add_event(EventSet, PAPI_TOT_INS)
/* Start counting */
    PAPI_start(EventSet)
/* reading the counters */
    PAPI_read(EventSet, values)
/* Stop counting */
    PAPI_stop(EventSet, values)
```

Check function return value
Example Program

```c
#include <papi.h>
int main()
{
    int events[2] = {PAPI_L2_TCM, PAPI_TOT_INS}, ret;
    long_long values[2];
    /* Initialize the PAPI library */
    ret = PAPI_library_init(PAPI_VER_CURRENT);
    if (ret != PAPI_VER_CURRENT) {
        fprintf(stderr, "PAPI library init error!\n");
        exit(1);}
    if ((ret = PAPI_start_counters(events, 2)) != PAPI_OK) {
        fprintf(stderr, "PAPI failed to start counters: %s\n", PAPI_strerror(ret));
        exit(1); }
    ...
    computation...
    if ((ret = PAPI_read_counters(values, 2)) != PAPI_OK) {
        fprintf(stderr, "PAPI failed to read counters: %s\n", PAPI_strerror(ret));
        exit(1); }
}
```

(Just) Timers

• A basic tool that is often very useful is (just) a timer
• Multiple types of timers are available
  • Make sure you use one with low overhead and high resolution
  • getrusage()
  • Beware of the difference between clock ticks and time: dynamic variation due to turbo boost and DVFS
  • Check overhead and resolution of timers by calling them repeatedly in a loop
Example Code for Timing Timers

Definition of a timer based on gettimeofday

double get_clock() {
    struct timeval tv; int ok;
    ok = gettimeofday(&tv, (void *) 0);
    if (ok<0) { printf("gettimeofday error"); } return (tv.tv_sec * 1.0 + tv.tv_usec * 1.0E-6);
}

t0 = get_clock();
for (i=0; i<N; i++) times[i] = get_clock();
t1 = get_clock();
printf("time per call: \%f ns\n", (1000000000.0*(t1-t0)/N));
Performance Analysis: profiling with gprof

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Profiling

• The basic idea of profiling is to collect data on execution characteristics of blocks of code
  • The easiest to identify block is a function

• Profile has come to mean something specific:
  • Data accumulated across time
  • Ignores time-series i.e. evolution of execution characteristics with time
  • Instead focuses on average time spent in each code block, for example
  • Saves time and creates manageable amount of data

• gprof is one of the oldest and still very useful tool
How to Use gprof

• Link your program with gprof libraries
  • Use "-pg" flag at link time

• Execute program normally, which produces some data in a binary file that you don’t need to look at
  • Called gmon.out

• In the same folder where the executable is (and the gmon.out file is)
  • Run “gprof <name-of-your-executable>”

• This will produce profile output
Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>time</th>
<th>seconds</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>48.46</td>
<td>3.15</td>
<td>3.15</td>
<td>38883398</td>
<td>0.00</td>
<td>0.00 TraversalWorker::work(Node&lt;ForceData&gt;*)</td>
</tr>
<tr>
<td>41.99</td>
<td>5.87</td>
<td>2.73</td>
<td>16213287</td>
<td>0.00</td>
<td>0.00 TraversalWorker::work(ExternalParticle*)</td>
</tr>
<tr>
<td>3.70</td>
<td>6.11</td>
<td>0.24</td>
<td>126598</td>
<td>0.00</td>
<td>0.05 Traversal&lt;ForceData&gt;::topDownTraversal(Node&lt;ForceData&gt;<em>, CutoffWorker&lt;ForceData&gt;</em>, State*)</td>
</tr>
<tr>
<td>1.54</td>
<td>6.21</td>
<td>0.10</td>
<td>12784</td>
<td>0.01</td>
<td>0.01 CkVec&lt;Particle&gt;::bubbleSort(int, int)</td>
</tr>
<tr>
<td>0.92</td>
<td>6.27</td>
<td>0.06</td>
<td>126648</td>
<td>0.00</td>
<td>0.00 MomentsWorker::work(Node&lt;ForceData&gt;*)</td>
</tr>
<tr>
<td>0.54</td>
<td>6.31</td>
<td>0.04</td>
<td>21133570</td>
<td>0.00</td>
<td>0.00 State::nodeOpened(unsigned long, Node&lt;ForceData&gt;*)</td>
</tr>
<tr>
<td>0.46</td>
<td>6.34</td>
<td>0.03</td>
<td>50</td>
<td>0.60</td>
<td>0.60 DataManager::hashParticleCoordinates(OrientedBox&lt;float&gt;&amp;)</td>
</tr>
<tr>
<td>0.46</td>
<td>6.37</td>
<td>0.03</td>
<td>38883398</td>
<td>0.00</td>
<td>0.00 State::nodeEncountered(unsigned long, Node&lt;ForceData&gt;*)</td>
</tr>
</tbody>
</table>

0.00  6.49  0.01  Request::deliverNode() |
0.00  6.49  0.00  1755170 | 0.00  0.00 TraversalWorker::bucketDone(unsigned long) |
0.00  6.49  0.00  1755170 | 0.00  0.00 State::bucketComputed(Node<ForceData>*, unsignaled long) |
0.00  6.49  0.00  253796 | 0.00  0.00 std::deque_base<Node<ForceData>*>, std::allocator<Node<ForceData>*>::_M_initialize_map(unsigned long) |
0.00  6.49  0.00  131462 | 0.00  0.00 State::nodeDiscarded(unsigned long, Node<ForceData>*) |
0.00  6.49  0.00  128297 | 0.00  0.00 CompareKeys(void*, unsignaled long) |
0.00  6.49  0.00  126648 | 0.00  0.00 std::_Rb_tree<unsigned long, std::pair<unsigned long const, Node<ForceData>*>, std::_Select1st<std::pair<unsigned long const, Node<ForceData>*>::_M_get_insert_hint_unique_pos(std::_Rb_tree_const_iterator<Node<ForceData>*, unsigned long const, Node<ForceData>*>::_M_get_insert_hint_unique_pos(Node<ForceData>*)*)*)*, std::less<unsigned long>, std::allocator<Node<ForceData>*>::_M_get_insert_hint_unique_pos(std::_Rb_tree_const_iterator<Node<ForceData>*, unsigned long const, Node<ForceData>*>::_M_get_insert_hint_unique_pos(Node<ForceData>*)*)*, unsignaled long const) |
0.00  6.49  0.00  64382 | 0.00  0.00 findSplitters(Particle*, int, int*, unsigned long*, int) |
0.00  6.49  0.00  63349 | 0.00  0.00 MomentsWorker::setLeafType(Node<ForceData>*) |
Call Graph View

```
-----------------------------------------------
info about calling Function 1

Info about calling function 2
...

foo : timing info about foo

Info about function g1 called by foo

Info about function g2 called by foo
...
-----------------------------------------------
-----------------------------------------------
0.00  0.00   1100/10000
TreePiece::doLocalGravity(RescheduleMsg*) [4]

0.00  0.00   1100/10000
TreePiece::doRemoteGravity(RescheduleMsg*) [5]

[2] 95.7  0.24   5.97  126598
Traversal<ForceData>::topDownTraversal(......)[2]

3.15  0.10 38883398/38883398
TraversalWorker::work(Node<ForceData>*) [3]

2.73  0.00 16213287/16213287
TraversalWorker::work(ExternalParticle*) [6]

0.00  0.00 1755170/1755170
State::bucketComputed(......) [1448]

0.00  0.00 1755170/1755170
TraversalWorker::bucketDone(unsigned long) [1447]

0.00  0.00 253196/253796
std::_Deque_base<Node<ForceData>*>,
std::allocator<Node<ForceData>*>* >::_M_initialize_map(unsigned long) [1449]
```

---
Utility of gprof and profiling

• Gives a quick view of where program’s execution time went
• Allows you to focus your attention on a smaller fraction of code that is responsible for a large fraction of execution time
  • At the level of functions
• It does not show which function instances or objects are costly
  • Since it shows total and average execution time spent in each function
• It does not show how time spent in specific function changes over time
• Since its based on sampling, there can be errors of estimation, especially for functions with tiny execution times