The 2009 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 11-15, 2009 St. Louis, USA

Improving Particle Filter Performance Using SSE Instructions

Peter Djeu and Michael Quinlan and Peter Stone

Abstract— **Robotics researchers are often faced with real-time constraints, and for that reason algorithmic and implementation-level optimization can dramatically increase the overall performance of a robot. In this paper we illustrate how a substantial run-time gain can be achieved by taking advantage of the extended instruction sets found in modern processors, in particular the SSE1 and SSE2 instruction sets. We present an SSE version of Monte Carlo Localization that results in an impressive 9x speedup over an optimized scalar implementation. In the process, we discuss SSE implementations of atan, atan2 and exp that achieve up to a 4x speedup in these mathematical operations alone.**

I. INTRODUCTION

In robotics, it is often critical to process sensory data or to compute decisions in real-time, that is to say at the rate at which sensory data is captured. For example, a robot with a camera that captures frames at 30 Hz should complete its sense-act loop at the same rate or risk missing sensing opportunities. Sometimes this constrains the processing algorithms that can be used (e.g. Hough transforms may be ruled out entirely) and sometimes it constrains the quality of the processing (e.g. the number of particles that can be processed in a particle filtering algorithm). As a result, an important component of robotics is code streamlining and optimization. A common source of untapped potential for code optimization is the use of the vector unit on CPUs to perform efficient and parallel computation.

Since 1998[1] most modern processors, including desktop and notebook/netbook processors, can perform SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) operations which allow a single instruction stream to drive parallel computation. In particular both Intel and AMD support the SSE (**S**treaming **S**IMD **E**xtension) instruction set and the follow-up instruction set SSE2 via an SSE vector unit [1]. Using the SSE vector unit allows the CPU to perform *4-wide* operations in place of *1-wide* operations; four additions can be performed in the time it usually takes to perform one. Under ideal settings, an SSE vector implementation can be up to 4x faster than the scalar implementation, and in cases such as the ones described in this paper, it can be even faster.

However, developing code that effectively uses the SSE instruction set has generally been restricted to graphics researchers or to the developers of heavily optimized libraries for specific tasks such as vision [2] and linear algebra [4]. While these libraries provide excellent performance on their intended tasks, the average roboticist has failed to take advantage of the SSE vector unit in developing his or her code/algorithms.

In this paper we present ground-up SSE implementations of key functions required in the robotics domain (atan, atan2, exp) and apply them to a commonly used and practical robotics algorithm: **M**onte **C**arlo **L**ocalization (MCL) [3]. In our implementation we achieve an impressive 9x speedup, surpassing the ideal speedup of 4x.

While this improvement in and of itself is noteworthy, one of the main objectives of this paper is to encourage and also instruct the robotics community on how to implement its own algorithms using the SSE instruction set. To help facilitate this task we are releasing the full source code for our MCL implementation, including the SSE components.

The remainder of this paper is structured as follows: Section II provides an overview of the basics of SSE. Section III introduces the Monte Carlo Localization algorithm and steps through the construction of the SSE version, with particular focus on the development of the SSE math extensions (such as atan2). Section IV will review and discuss the run-time performance results of both the overall MCL algorithm and the individual math operations. We will conclude and present future work in Section V.

II. USING THE SSE INSRUCTION SET

Traditional CPU instructions take single values as their operands. A typical floating point addition instruction has the following form, which we call *1-wide addition*.

```
float a = 1.0f;
float b = 5.0f;
float c = a + b;  // c: 6.0f
```

In SSE, the data types are expanded from ‘float’ to ‘sse4Floats’[2], which include the values of 4 separate floating points. When an addition is performed on two SSE operands, each of the 4 elements is independently summed. In other words, element 0 in the first operand is added to element 0 in the second operand, element 1 in the first operand is added to element 1 in the second operand, and so on. We will refer to this operation as a *4-wide addition*.

In the following example, we will use a C++ constructor for sse4Floats that takes 4 floating point values. The ‘+’ operator is overloaded so that if both operands are sse4Floats, the SSE unit is invoked to perform a 4-wide addition.

```
sse4Floats a = sse4Floats(1.0f, 2.0f, 3.0f, 4.0f);
sse4Floats b = sse4Floats(5.0f, 6.0f, 7.0f, 8.0f);
sse4Floats c = a + b;  // c: (6.0f, 8.0f, 10.0f, 12.0f)
```

All authors are in the Department of Computer Science, The University of Texas at Austin. djeu,mquinlan,pstone@cs.utexas.edu

[1]In 1998 AMD released the K6-2 processors, this was followed in 1999 by Intel’s Pentium III processors

[2]The built-in SSE data type is actually ‘__m128’, however we encapsulate this type into a C++ class, which we call ‘sse4Floats’.

978-1-4244-3804-4/09/$25.00 ©2009 IEEE

We will use the term *1-wide* and *4-wide* to refer to mathematical operations that are performed analogously to the addition operations. The term *scalar* will be used interchangeably with 1-wide and *vector* interchangeably with 4-wide.

Apart from 4-wide floating point addition, the SSE vector unit also provides support for other arithmetic operations, such as subtraction, multiplication, division, minimum, maximum, and bitwise operations. The vector unit supports vectors of four 32-bit integers (aka ints) and has built-in support for addition, subtraction, and bitwise operations. Processors which support later versions of SSE also provide multiplication, minimum, and maximum operations for integers.

A. SSE Masks

One difficulty that occurs when using SSE is that a single instruction stream must be used for all processing. This is especially troubling when implementing algorithms that would normally require branching. Consider the following example, which returns the absolute value of f.

```
float abs(float f) { return (f >= 0.0f) ? f : -f; }
```

When implementing a 4-wide version of absolute value for an sse4Floats consisting of the values (-1.0f, -2.0f, +3.0f, +4.0f), the first 2 elements need to be negated, while the last 2 elements do not. However, we must use the same instruction stream on all elements as per the definition of SIMD.

The solution is to evaluate both sides of the branch and to mask out the result from one branch and mask in the result from the other. The two results are then combined and only the masked-in result survives. We now revisit the scalar implementation to support this branching paradigm using bitwise operations. For brevity, the implementation of the two reinterpretation helper functions are not shown.

```
int reint_f2i(float f); // reinterpret float as int
float reint_i2f(int i); // reinterpret int as float
float abs(float f) {
  int MASK_TRUE  = 0xffffffff;
  int MASK_FALSE = 0x00000000;
  int mask = (f >= 0.0f) ? MASK_TRUE : MASK_FALSE;
  int true_result  =  mask & reint_f2i( f);
  int false_result = ~mask & reint_f2i(-f);
  int blend_result =  true_result | false_result;
  return reint_i2f(blend_result);
}
```

As you will notice, this version of absolute value still contains one branch when selecting the value of the variable 'mask', the value being either a 32-bit int with all bits set (the *true mask*) or a 32-bit int with all bits unset (the *false mask*).

The SSE vector unit solves this problem by mandating that all of its comparison operations produce either a true mask or a false mask in each element of the 4-wide result. In the following example, we generate an *SSE mask* by comparing the input value to another sse4Floats set completely to zero. We implement SSE masks using the sseMask data type, which is a C++ class that encapsulates a built-in SSE data type representing a 4-wide mask.

```
sse4Floats a = (1.0f, 0.0f, -1.0f, -2.0f);
sse4Floats b = (0.0f, 0.0f,  0.0f,  0.0f);
sseMask mask = (a >= b);
// mask: (MASK_TRUE, MASK_TRUE, MASK_FALSE, MASK_FALSE)
```

This sseMask can then be used in a 4-wide blend operation which takes three operands: an sseMask, a true operand, and a false operand. The return value is an sse4Floats that is set to the value of the true operand at all elements where the sseMask had value MASK_TRUE. The return value is set to the false operand at all other locations. The operators '&', '|', and '∼' are overloaded to invoke the corresponding bitwise operations in the SSE unit.

```
sse4Floats blend4 (sseMask mask, sse4Floats in_true,
                                 sse4Floats in_false) {
  return (mask & in_true) | (~mask & in_false);
}
```

The following is an SSE implementation of absolute value. The '>=' operator is overloaded to invoke the greater-than-or-equal-to comparison operator in the SSE unit, while the '-' operator is overloaded to invoke the unary negation operator in the SSE unit.

```
sse4Floats abs(sse4Floats f) {
  sse4Floats zeros = sse4Floats(0.0f, 0.0f, 0.0f, 0.0f);
  sseMask is_non_neg = (f >= zeros);
  return blend4(is_non_neg, f, -f);
}
```

To summarize, masking allows us to conditionally perform operations on only certain elements within an sse4Floats rather than on all elements. Masking comes with a cost, however, since both the true input and the false input must be evaluated whenever a mask is used. In scalar code, work can often be saved by descending down only one branch of a conditional, while in pure SSE code both sides of the branch must always be taken, even in cases where the sseMask consists of 4 MASK_TRUE's or 4 MASK_FALSE's.

B. Reduction Operations

It is often necessary to perform an operation which reduces the 4 values within an sse4Floats to a single floating point value.

```
sse4Floats a = sse4Floats(1.0f, 2.0f, 3.0f, 4.0f);
float      b = reduce_add(a);  // b: 10.0f
```

These operations which work internally within a single sse4Floats are known as *reduction operations*. Reduction operations are typically slower than their 4-wide counterparts; however they are usually needed only sparingly. For example, the following block of code returns the average value of all floats in an array of sse4Floats. Notice that only one reduction operation is needed.

```
//n - the number of 4-wides in arr
float getAverageValue(sse4Floats *arr, int n) {
  sse4Floats accum = sse4Floats(0.0f, 0.0f, 0.0f, 0.0f);
  for (int i = 0; i < n; i++)
    accum = accum + arr[i];     // 4-wide add
  float total = reduce_add(accum);
  return total / (n * 4.0f);
}
```

C. SSE Performance

Using the SSE vector unit to perform 4-wide operations in place of 1-wide operations allows us to perform four arithmetic operations in the time it would take to perform one. Under ideal settings, an SSE implementation can be up to 4x faster than the scalar implementation. We will refer to the ratio of the scalar version's run-time to the SSE version's run-time as *parallel speedup*.

There are many reasons why parallel speedup falls below 4x. Often, reformatting data to conform to an SSE-compatible layout is expensive, as is converting from SSE format to an output format. More generally, we must operate within the constraints of Amdahl's Law, where any part of an algorithm that cannot be parallelized will limit the overall speedup. In applications where data is not readily available in multiples of 4, some elements in the SSE vector will be left empty, which further limits speedup. Finally, for algorithms that have *deep conditional branches*, where one or both sides of the branch contain a significant amount of work, the SSE version will suffer because it must perform the work on both sides of the branch, compared to the scalar version which only needs to execute the taken branch.

It is possible, however, to achieve a parallel speedup that is greater than the ideal speedup of 4x, which is evidenced in this work. The reason this is possible is because of a fundamental difference in the hardware that evaluates SSE code and scalar code which exists on many platforms. For example, on many Intel chips the ALU that performs SSE floating point operations is faster than the ALU which handles scalar floats since the latter is based on the aging design of the x87 math co-processor. In other words, a parallel speedup that exceeds 4x is possible due to a combination of good data parallelism (SIMD) and intrinsic speedup due to using better hardware (the SSE ALU is faster than the scalar ALU).

In our experience, the best time to switch from a scalar implementation to an SSE implementation is when the scalar algorithm requires performing arithmetic operations independently but identically over a large data set. For example, an algorithm that evaluates the exp(x) function on every element of an array is a prime candidate for conversion to SSE.

III. SSE AND MONTE CARLO LOCALIZATION

A. Monte Carlo Localization (Particle Filters)

For the purpose of this paper we will describe the implementation of a simple version of Monte Carlo Localization (MCL) [3]. We hope that our description provides others with the necessary framework to begin including SSE in deployable real-world implementations. We have chosen to keep the example as straightforward as possible (see Algorithm 1) in an attempt to clearly demonstrate the modifications needed to use the SSE instruction set. Notice that in Line 10 of our implementation, we compute particle likelihoods by accumulating the exponent of the probability in the inner loop and perform the exponentiation later in the estimatePose() helper function (Line 13). Both versions of the MCL algorithm (scalar and SSE) benefit from this optimization.

Our MCL implementation will be taking observations (measurements) in the form of the distance (mm) and bearing (radians) to known reference objects. Figure 1 presents the output of the algorithm when either one or three objects are being observed.

Algorithm 1 Particle Filter Localization

1: **for all** $p \in$ particles **do**
2: p.exponent $\leftarrow 0$
3: **end for**
4: **for all** $o \in$ observations **do**
5: **for all** $p \in$ particles **do**
6: $\text{dist}_{expected} \leftarrow$ getDistanceToPoint(p.pos,o.pos)
7: $\text{bear}_{expected} \leftarrow$ getBearingToPoint(p.pos,o.pos)
8: $\text{pe}_{dist} \leftarrow$ getDistanceSimExponent($\text{dist}_{expected}$,o.dist)
9: $\text{pe}_{bear} \leftarrow$ getBearingSimExponent($\text{bear}_{expected}$,o.bear)
10: p.exponent $\leftarrow$ p.exponent + pe_{dist} + pe_{bear}
11: **end for**
12: **end for**
13: estimatePose(particles)

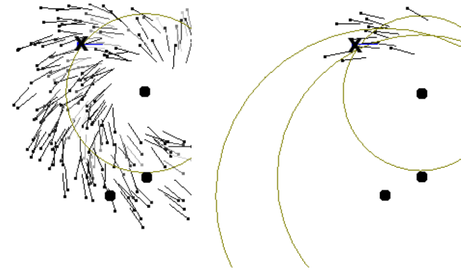

Fig. 1. Example Particles: In the *left* image, the robot (drawn as an X) observes a single object. One reference object is not enough for localization, which results in the most likely particles forming a ring around that object. In the image on the *right*, the robot is observing all three objects, resulting in the most likely particles clustering around the robot's true location.

B. SSE Math Extensions

The processor's built-in support for SSE, which in this paper is considered to be the SSE instruction sets 'SSE1' and 'SSE2', is sufficient for implementing many algorithms [1]. We choose to limit our scope to these two instruction sets in order to support older processors. For Monte Carlo localization, we need to evaluate atan2(y, x) and exp(x) for 4-wide inputs and return results that are also 4-wide. Neither function is part of the built-in instruction set so we will provide details on our implementations.

C. SSE Data Structures

For our SSE implementation, we use SSE data structures to represent four particles at a time. Our 4-wide particle data structure represents 4 separate particles, where each particle consists of an x coordinate, a y coordinate, an orientation (aka bearing), a probability exponent, and a probability.

```
class Particle_4Wide {
    sse4Floats x;          // 4-wide x coordinate
    sse4Floats y;          // 4-wide y coordinate
    sse4Floats o;          // 4-wide bearing
    sse4Floats exponent;   // prob exponent
    sse4Floats prob;       // equal to exp(exponent)
}
```

We reuse the basic outline for a particle filter from Algorithm 1 for our SSE implementation, where we overload our functions from our scalar version to recognize and return an sse4Floats in place of a float and a Particle_4Wide in place of a Particle.

Another modification we need for the SSE particle filter is to expand the observations. Normally, observations and particles are both 1-wide. However, now that particles are 4-wide, the observation must also be 4-wide. This is accomplished by duplicating the data from one observation across all elements of a 4-wide in a process called *expansion*. The expanded observation can now be compared against the 4-wide particle. To implement expansion, we will use the function, expand(float x), which creates an sse4Floats from a single float.

D. Bearing To A Point

Calculating the expected bearing from a single particle to an object (Line 4 in Algorithm 1) requires the following code. The normalizeAngle() function converts an angle from the interval (-∞, ∞) to the interval [-π, π].

```
AngRad getBearingToPoint(Particle particle, Point2D obj) {
  float dx = obj.x - particle.x;
  float dy = obj.y - particle.y;
  AngRad theta = atan2(dy, dx);
  return normalizeAngle(theta - particle.o);
}
```

Notice that we need to perform an atan2 during the bearing computation for *every* combination of observation and particle. To implement atan2(y, x), we will first implement the helper function atan(x). In the following sections, we describe our SSE implementations for both of these functions.

1) Function 1: atan: The atan(x) function determines the angle theta on [-π/2, π/2] for which tan(theta) = x. Our implementation computes atan by evaluating the first seven terms in Euler's power series for atan. This series expansion converges more rapidly than the more commonly used power series for atan. The helper method __atan_rd(x) evaluates the actual power series on the **r**educed **d**omain (rd) of [0, 1], which guarantees that the power series will converge. In order to convert a generic input to __atan_rd(x)'s reduced domain, we use the following two trigonometric identities:

```
Trig Ident 1: atan(x) = -atan(-x)
Trig Ident 2: atan(x) = PI/2 - atan(1/x)
```

These two identities can be used to transform any input x from the real number line to the interval [0, 1]. We will call this process of compressing inputs to a smaller interval *domain reduction*. When domain reduction is used, it usually has a corresponding *range expansion*. In this case, we first apply domain reduction in atan() in preparation for the call to __atan_rd(x). After __atan_rd(x) is evaluated, we use range expansion to fix the sign of the return value from __atan_rd(x) if we used the first trigonometric identity during domain reduction. The second part of range expansion is to subtract from π/2 if we used the second trigonometric identity during domain reduction. We provide a further optimization by subtracting from either -π/2 or π/2 to guarantee that the output from atan(x) is in the range [-π/2, π/2].

In order to improve the accuracy of the __atan_rd(x) function, we provide a small fix for values of x that are near 0. These values will undergo floating point underflow during self-multiplication, which drives the output of __atan_rd(x) to zero. To alleviate this problem, we simply return the value of x for all values for which the reference implementation of atan() from math.h also returns the value x. The largest such value of x is about 0.000352.

```
//domain: [0.0, 1.0], range: [-PI/2, PI/2]
sse4Floats __atan_rd(sse4Floats x) {
  sse4Floats ps; // contains result of power series
  // ...evaluation of power series omitted...
  //fix values that generate 0 but should just be x
  sse4Floats thresh  = expand(0.000352f);
  return blend4(x < thresh, x, ps);
}
```

```
//domain: [-INF, INF], range: [-PI/2, PI/2]
sse4Floats atan(sse4Floats x) {
  sse4Floats zero = expand(0.0f);
  sse4Floats one  = expand(1.0f);
  sse4Floats pi_over_two = expand(M_PI * 0.5f);
  //take absolute value
  sseMask neg_x = x < zero;
  sse4Floats sign_conv = blend4(neg_x, -one, one);
  sse4Floats abs_x = sign_conv * x;
  //invert all values that are greater than one
  sseMask needs_invert = (abs_x > one);
  sse4Floats inv_abs_x = one / abs_x;
  sse4Floats x_rd = blend4(needs_invert, inv_abs_x, abs_x);
  sse4Floats raw_atan = __atan_rd(x_rd);
  //fix signs based on the signs of the input
  sse4Floats theta = sign_conv * raw_atan;
  //correct output range by subtracting from PI/2 or -PI/2
  sse4Floats cons = blend4(neg_x,-pi_over_two, pi_over_two);
  return blend4(needs_invert, cons-theta, theta);
}
```

2) Function 2: atan2: The atan2(y, x) function is a more robust version of atan(x). The atan2(y, x) function returns the angle theta on [-π, π] that is formed by the positive x axis and the line segment connecting the origin to the point (x, y). The atan2(y, x) function is used during bearing computations in the particle filter.

Our implementation uses atan(x) as a helper method by computing y/x and then invoking atan(x). The raw result from atan(x) is then put through a range expansion based on the quadrant in which the input to atan2(y,x) lies. The range expansion takes the return value of atan(x) and converts it from either quadrant 4 to quadrant 2 by adding π or from quadrant 1 to quadrant 3 by subtracting π, as necessary.

```
//returns MASK_TRUE if element has its sign bit set,
//returns MASK_FALSE otherwise
sseMask is_neg_special(sse4Floats x);

//domain [-INF, INF]x[-INF, INF], range: [-PI, PI]
sse4Floats atan2(sse4Floats y, sse4Floats x) {
  sse4Floats pi = expand(M_PI);
  sse4Floats theta = atan(y / x);
  //treat -0 as though it were negative
  sseMask neg_x = is_neg_special(x);
  sseMask neg_y = is_neg_special(y);
  //move from quadrant 4 to 2 by adding PI/2
  sseMask in_q2 = neg_x & ~neg_y;
  sse4Floats q2f = blend4(in_q2, theta+pi, theta);
  //move from quadrant 1 to 3 by subtracting PI/2
  sseMask in_q3 = neg_x & neg_y;
  return blend4(in_q3, theta-pi, q2f);
}
```

E. Distance/Bearing Similarity

Typically, the distance and bearing similarity are computed by evaluating an exponential e^x for each similarity within the inner loop of the particle filter. These similarities are then accumulated into a total likelihood for the particle via multiplication. However, we can reduce the number of exponentiations that are needed by deferring the exponentiation until pose estimation for the robot. To do so, we simply sum the exponents of the similarity values within the inner loop of the particle filter (Lines 8–10 in Algorithm 1). Later, we produce a mathematically identical result by evaluating the exponential in the call to the estimatePose() helper function (Line 13 of Algorithm 1). Note the use of exp(x) in Line 4 of Algorithm 2.

Algorithm 2 estimatePose() function

1: $pose_{accum} \leftarrow 0$
2: $prob_{accum} \leftarrow 0$
3: **for all** $p \in$ particles **do**
4: p.prob ← exp(p.exponent)
5: $pose_{accum} \leftarrow pose_{accum}$ + p.pose * p.prob
6: $prob_{accum} \leftarrow prob_{accum}$ + p.prob
7: **end for**
8: $pose_{mean} \leftarrow pose_{accum}$ / $prob_{accum}$
9: $pose_{stddev} \leftarrow$ computeStdDev(particles, $pose_{mean}$)

By deferring the evaluation of exp(x) until estimatePose(), we can reduce the number of exponentiations needed by a factor of N, where N is the number of *observations* from Algorithm 1. In the remainder of this section we describe our SSE implementation of exp(x).

1) Function 3: exp: The exp(x) function evaluates e^x. The exp(x) function is used for computing likelihoods in the particle filter. Our implementation for exp(x) is broken down into three helper methods, __exp_rd(x) and its helper methods __exp_exponent(ipart) and __exp_mantissa(x).

The first step in the exp(x) function is to pass its input to __exp_rd(x), which works on the reduced domain of approximately $(-87.3, 88.7)$. Inputs outside of this domain are trivial to evaluate and will be handled by the main exp(x) function later.

The __exp_rd(x) function evaluates e^x by evaluating the result in two parts based on the composition of a 32-bit float. Recall that a 32-bit float consists of 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa. The first part of our algorithm determines the correct 8 bits for the exponent in the helper function __exp_exponent(ipart). The second part of our algorithm determines the correct 23 bits for the mantissa in the helper function __exp_mantissa(x). These results are combined and returned by the function __exp_rd(x). Finally, for any input that exceeds the reduced domain of __exp_rd(x), the main exp(x) function masks in either 0.0 or ∞ based on whether the input is less than or greater than the reduced domain, respectively.

It should be noted that __exp_mantissa(x) evaluates e^x on the reduced domain $[0.0, log_2(e)]$ based on a special seven-term power series that has been fitted for high accuracy on the reduced domain. The coefficients differ from the ones used in the standard series expansion for e^x so we refer the reader to our code release.

The following code uses the new data type sse4Ints. This data type is a C++ class that encapsulates an '__m128i', which represents four 32-bit integers rather than four 32-bit floats. For this class we have overloaded the << operator to invoke the SSE unit's bitwise left shift instruction.

```
// domain: [-126, 127], range: [2^-126, 2^127]
sse4Floats __exp_exponent(sse4Ints ipart) {
  sse4Floats one = expand(1.0f);
  sse4Ints fasi = (ipart << 23) + reint_f2i(one);
  return reint_i2f(fasi);
}
//domain: [0.0, log_2(e)], range:  [1.0, 2.0)
sse4Floats __exp_mantissa(sse4Floats x) {
  sse4Floats ps; // contains result of power series
  // ...evaluation of power series omitted...
  return ps;
}
//domain: approx (-87.3, 88.7), range: [0, INF]
sse4Floats __exp_rd(sse4Floats x) {
  sse4Floats log_e2  = expand(0.693147f);
  sse4Floats log_2e  = expand(1.442695f);
  sse4Ints   ipart = cast_f2i(log_2e*x);
  sse4Floats fpart = x - log_e2*cast_i2f(ipart);
  return __exp_exponent(ipart)*__exp_mantissa(fpart);
}
```

```
//domain: [-INF, INF], range:  [0, INF]
sse4Floats exp(sse4Floats x) {
  sse4Floats raw_exp = __exp_rd(x);
  //fix values that are below or above a certain threshold
  sse4Floats zero     = expand(0.0f);
  sse4Floats infinity = expand(INF);
  sse4Floats min_thr  = expand(-87.336548f);
  sse4Floats max_thr  = expand( 88.722839f);
  sse4Floats min_fix  = blend4(x<=min_thr,zero,raw_exp);
  return blend4(x>=max_thr,infinity,min_fix);
}
```

IV. RESULTS

Three experiments were performed to test various aspects of run-time performance. The first experiment tests how well both the scalar and SSE particle filters scale with the number of particles as well as their respective accuracy. The second experiment examines how many particles each version can support when there is a hard real-time constraint. The third experiment isolates the performance of the SSE versions of the advanced math operations. All experiments were conducted on a 2.66 GHz Core 2 Duo using the Intel Compiler version 10.1.029.

A. Scaling Experiment

The purpose of this experiment is to measure the run-time performance the SSE and scalar particle filters while varying the number of particles. Maximum optimization settings were used for both the scalar and SSE versions. These experiments were run with noise-free observations and with three observations at each measurement update. The results are shown in Table I and Figure 2.

We achieve a consistent parallel speedup of at least 7.6x in all of our runs, surpassing the ideal limit of 4x due to a combination of good data parallelization and use of better ALUs in the SSE version. This result provides strong motivation for using an SSE implementation over a scalar implementation.

As expected, localization error decreases as the number of particles increased. We reach a saturation point at ≥ 16384 particles, where additional particles no longer improve the accuracy of the pose estimate. Notice that for sufficiently low n, the particles are too sparsely distributed and all have 0 likelihood. When this occurs, the pose of the robot cannot be estimated and the error is infinite, which is represented by N/A in the table.

n	Scalar (s)	SSE (s)	Speed-up	Error	
				mm	rad
4	0.000023	0.000003	7.7	N/A	N/A
16	0.000028	0.000003	9.3	N/A	N/A
64	0.000047	0.000006	7.8	1590.0	0.513
256	0.000116	0.000015	7.7	820.0	0.307
1024	0.000403	0.000053	7.6	359.6	0.155
4096	0.001607	0.00020	7.7	171.0	0.0742
16384	0.006647	0.000831	8.0	109.0	0.0447
65536	0.031178	0.003308	9.4	108.0	0.0446
262144	0.125763	0.013664	9.2	110.0	0.0450
1048576	0.460547	0.057089	8.1	109.0	0.0446

TABLE I

SCALING RESULTS OVER n PARTICLES. ERROR IS THE AVERAGE ERROR OVER 100 RANDOM CONFIGURATIONS OF PARTICLES.

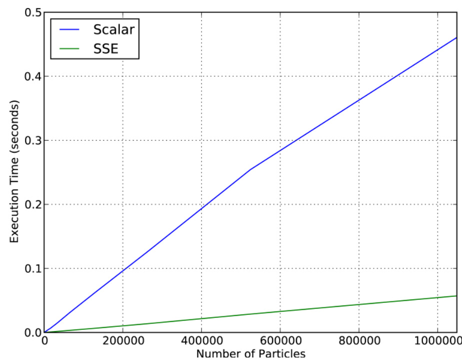

Fig. 2. Timing results for our scalar and SSE implementations of MCL.

B. Real-time Experiment

This experiment determines the number of observations we can process at real-time rates without ever going over a hard deadline of 30 Hz. We use 16384 particles, the saturation point determined in the scaling experiment. We vary the number of observations until we find the largest value that can meet the real-time deadline for 500 consecutive invocations of the particle filter, where each invocation is given a new set of observations.

The scalar version of the particle filter can handle up to 20 observations in this test, while the SSE version can handle 157 observations. The observed increase in particles by using SSE is a factor of 7.9x, which closely matches the speedup of 8.0x for 16384 particles from Table I.

C. Performance of SSE Math Functions

In this section we present the raw performance of the SSE math algorithms (atan, atan2 and exp). Table II summarizes the performance. Notice the exp() function approaches the optimal 4x speedup.

Function	Scalar (sec)	SSE (sec)	Speedup
atan	38.086196	20.303419	1.8769
atan2	65.351570	28.582232	2.2644
exp	19.501226	4.967288	3.9259

TABLE II

SSE MATH RESULTS: *exp* TESTS ALL FLOATS ON THE INTERVAL [-80.0, 80.0]. *atan* TESTS ALL FLOATS ON THE INTERVAL [$-\infty$, ∞]. *atan2* TESTS ALL PAIRS OF FLOATS ON THE UNIT CIRCLE.

V. CONCLUSION AND FUTURE WORK

This article described the construction of an SSE version of a Monte Carlo Localization algorithm, including developing SSE mathematical operations that are frequently required in the robotics domain. We have shown that a 9x speedup is possible over optimized scalar code. This speedup allows us to either a) process the same number of particles in less time or b) increase the accuracy of our localization by using a greater number of particles. The ability to support more particles also reduces the need for MCL algorithms to perform sophisticated and often expensive particle resampling.

Almost all processors on the market directly support SSE. Our results demonstrate that there is significant untapped potential in the SSE vector unit on these processors. To facilitate the exploration of this potential by the robotics community, we have developed and will open-source the implementation of our SSE particle filter and underlying components.

In future work we will extend our library to additional mathematical operations and sub-routines. We hope to roll out fully vectorized versions of common robotics algorithms and to take advantage of SSE in loop-heavy tasks such as processing lidar sensor readings.

We refer the reader to our source code release for additional implementation details not covered in this paper:

http://www.cs.utexas.edu/users/mquinlan/mcl_sse.html

ACKNOWLEDGEMENTS

We would like to thank Warren Hunt for contributing the design and implementation of the exp function. This research is supported in part by grants and funding from the National Science Foundation (CNS-0615104), DARPA (FA8750-05-2-0283 and FA8650-08-C-7812), the Federal Highway Administration (DTFH61-07-H-00030), General Motors. and Intel Corporation. The authors also thank Donald Fussell and Katherine Bontrager for comments and feedback on the paper.

REFERENCES

[1] Intel Corporation, Getting started with SSE/SSE2 for the Intel Pentium 4 Processor, White paper, http://www.developers.net/intelisnshowcase/view/116.

[2] Intel Open Source Computer Vision Library (OpenCV), http://opencvlibrary.sourceforge.net.

[3] Sebastian Thrun and Wolfram Burgard and Dieter Fox, Probabilistic Robotics (Intelligent Robotics and Autonomous Agents, The MIT Press, 2005.

[4] R. Clint Whaley and Jack Dongarra, Automatically Tuned Linear Algebra Software, Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999, Latest library at http://math-atlas.sourceforge.net/.