RED HAT :: CHICAGO :: 2009

SUMMIT

FOLLOW US:

TWITTER.COM/REDHATSUMMIT

TWEET ABOUT US:

ADD #SUMMIT AND/OR #JBOSSWORLD TO THE END OF YOUR EVENT-RELATED TWEET

presented by

RED HAT :: CHICAGO :: 2009
SUMMIT

Acceleration Through Stream Computing

Ulrich Drepper
Consulting Engineer, Red Hat
2009-9-4

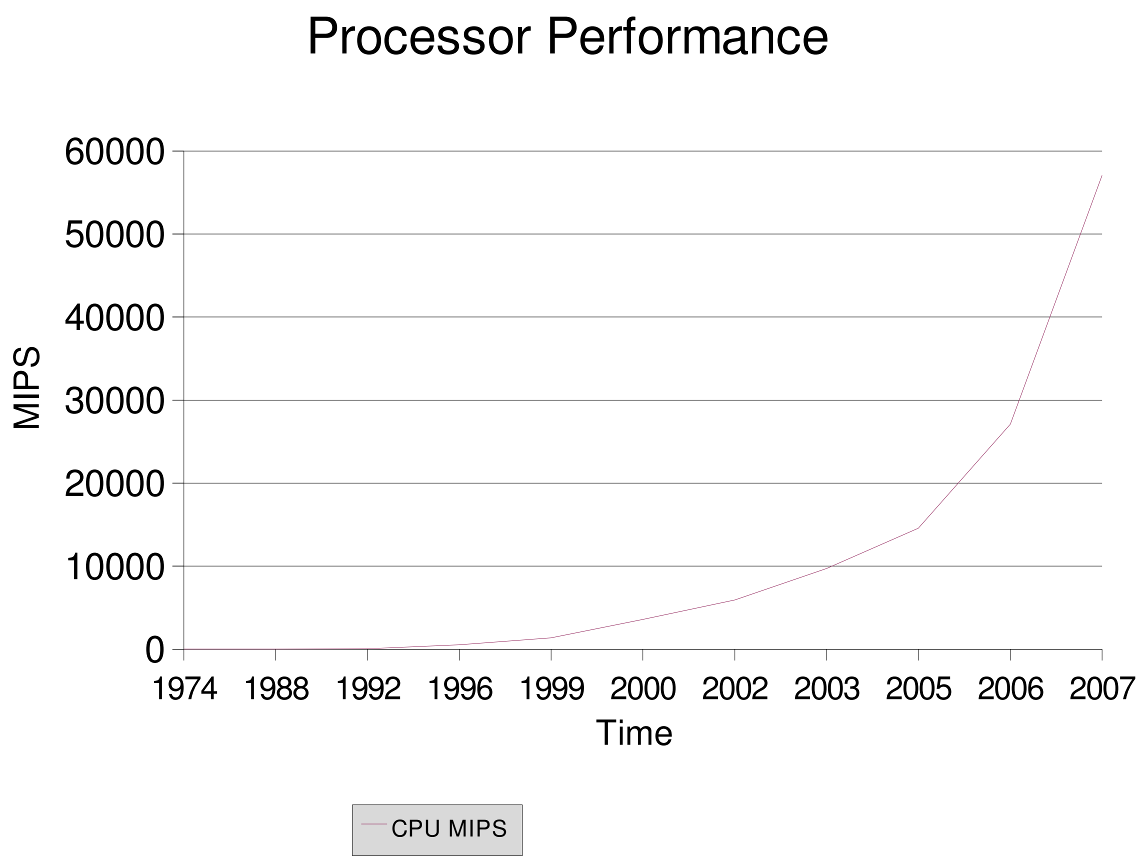
presented by

Overview

- Why Stream Computing?
- Simple Example
- Benchmarking
- Complex Examples
- Scalable Stream Programming Techniques

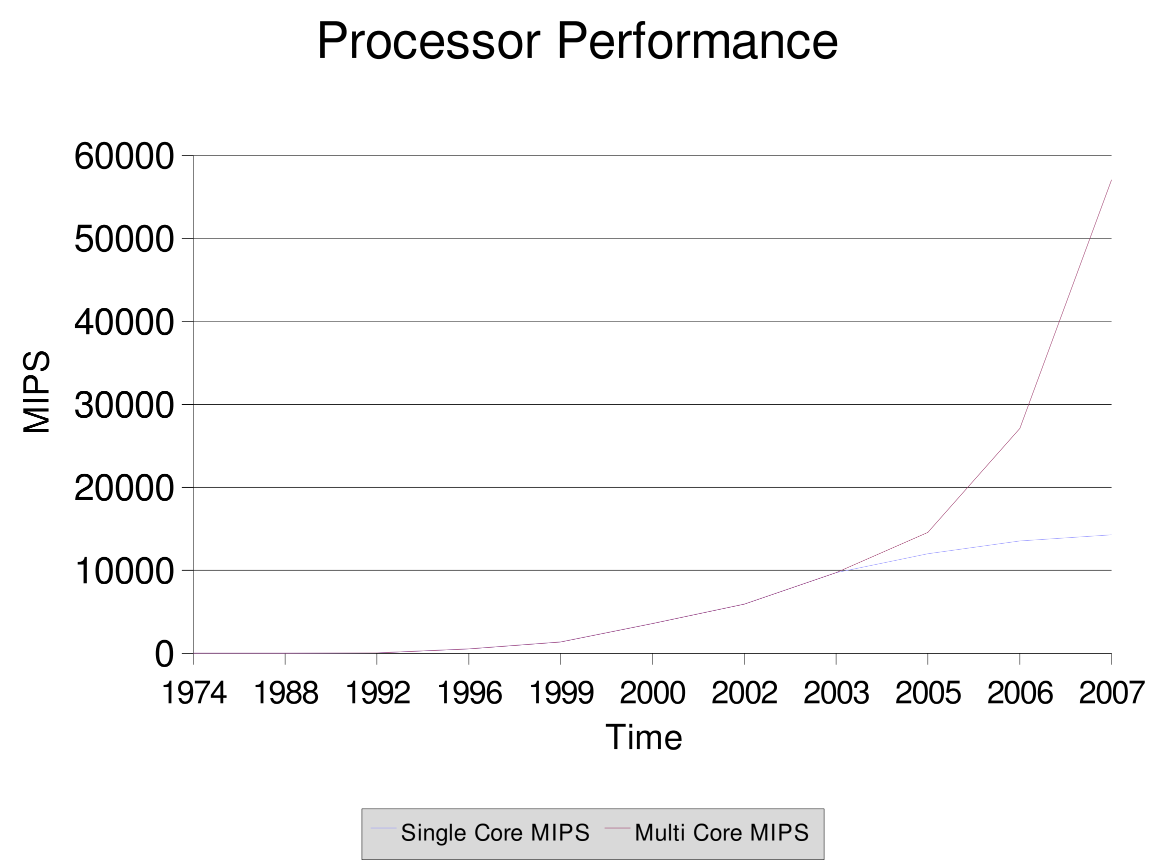
Why Stream Computing?

- What CPU manufacturers want you to believe:



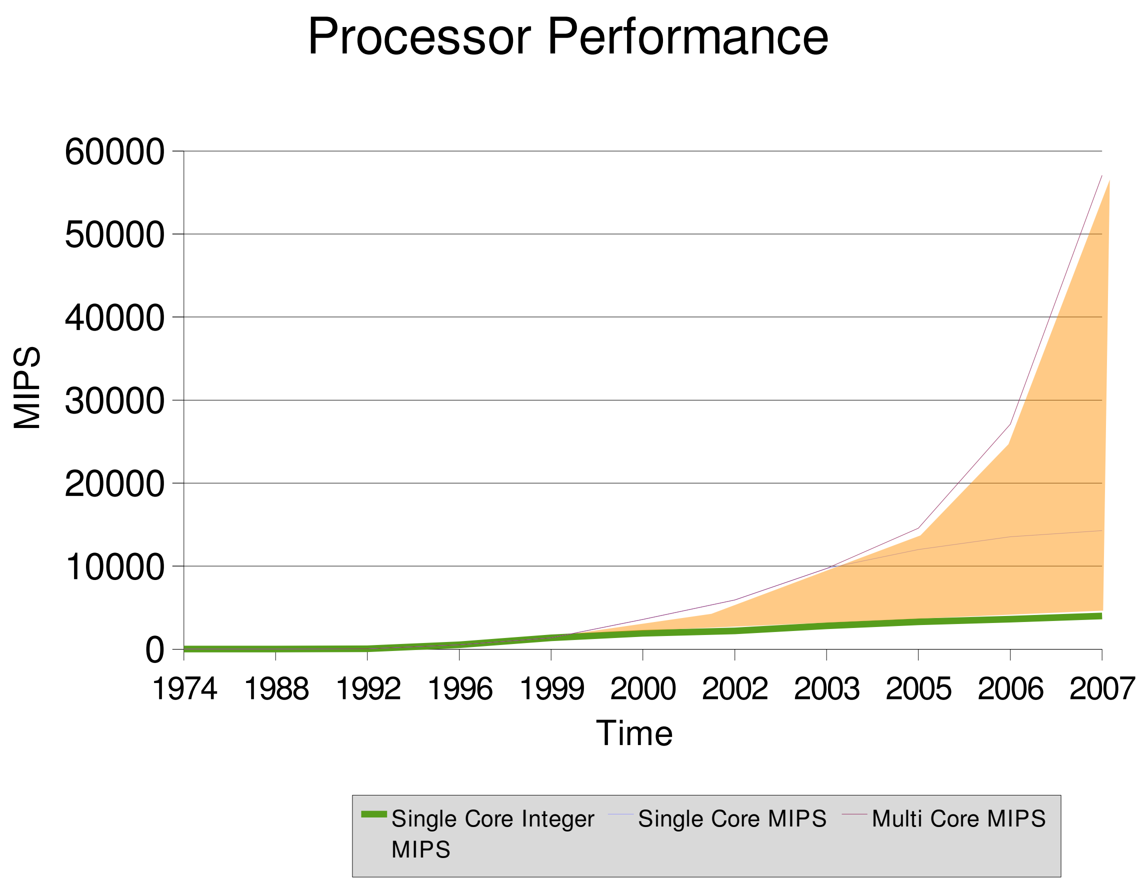
Why Stream Computing?

- We need parallelism:



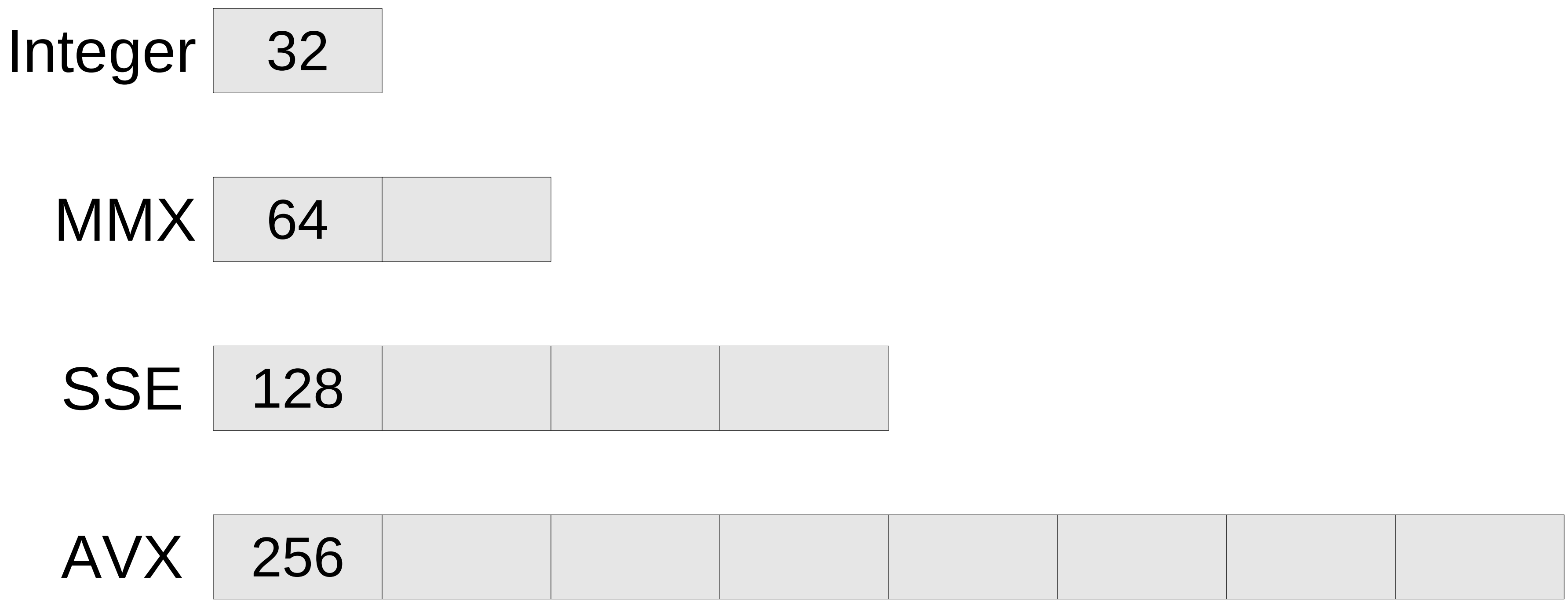
Why Stream Computing?

- … but also need Stream Computing



Why Stream Computing?

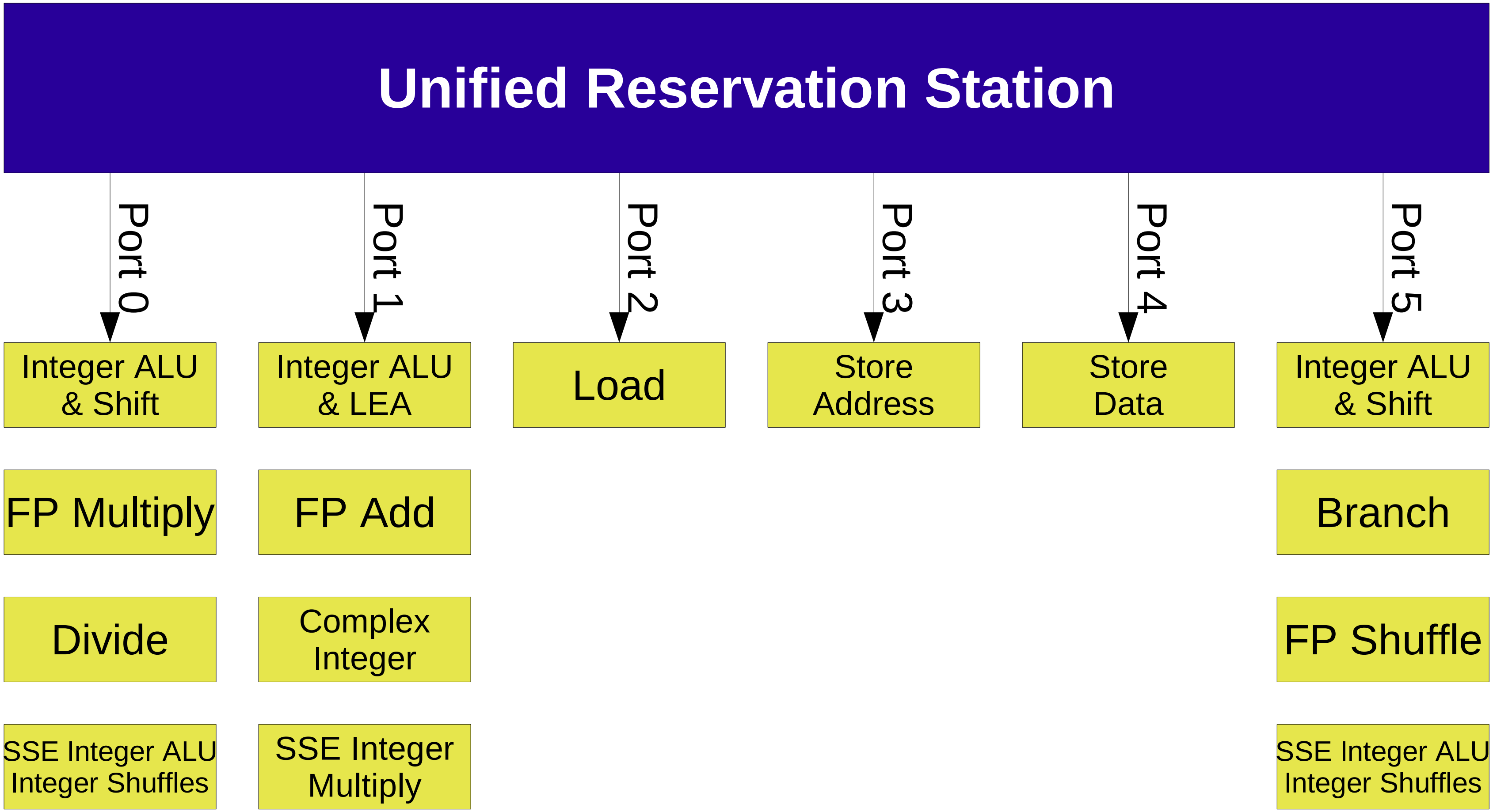
- Register width increases



And 512 bytes in future

Why Stream Computing?

- Modern Micro-Architecture (Nehalem):



Simple Example

- Prerequisite: data in arrays
- Simple: vector arithmetic

$$\vec{z} = \vec{x} * f + \vec{y}$$

- First Implementation:

```
extern float *x, *y, *z;
for (i = 0; i < N; ++i)
  z[i] = x[i] * f + y[i];
```

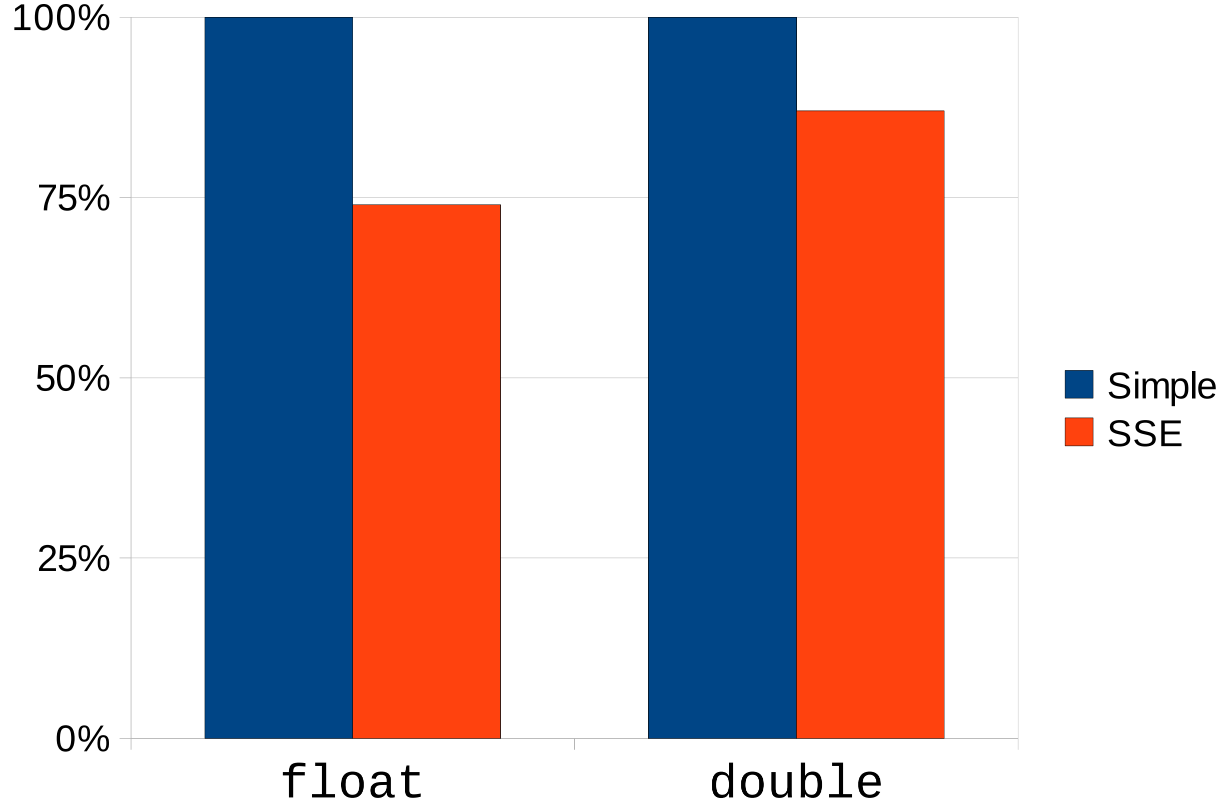
Simple Example

- Using SSE:

```
extern union { float f[N]; __v4sf v[N/4]; }
   *x, *y, *z;
__m128 vf = _mm_set_ps1(f);
for (i = 0; i < N / 4; ++i)
  z->v[i] = _mm_add_ps(_mm_mul_ps(x->v[i], vf),
                       y->v[i]);
```

Benchmarking

- Performance:



- Up to 25% faster

Complex Examples

- Today not only arithmetic stream instructions
  - Not only floating point, also integer
  - Complex move instructions
  - Logic instructions
  - Comparison instructions
  - Many more
- Complex control flow possible
- Even more support coming

Complex Examples

- Example using conditional:

```
void lscale(float *out, const float *in) {
  for (unsigned i = 0; i < N; ++i)
    if (in[i] > 10)
      out[i] = 10 + (in[i] – 10) * 9 / 10;
    else
      out[i] = src[i];
}
```

Complex Examples

- Using SSE2:

```
void lscale(__v4sf *out, const __v4sf *in) {
  __m128 v10 = _mm_set_ps1(10.0f), v09 = _mm_set_ps1(0.9f);
  for (unsigned i = 0; i < N / 4; ++i) {
    __m128 cmp = _mm_cmp_gt(in[i], v10);
    __m128 tmp = _mm_add_ps(v10, _mm_mul_ps(_mm_sub_ps(in[i], v10),
                                        V09));
    out[i] = _mm_or_ps(_mm_andnot_ps(cmp, in[i]),
                       _mm_and_ps(cmp, tmp));
}
```

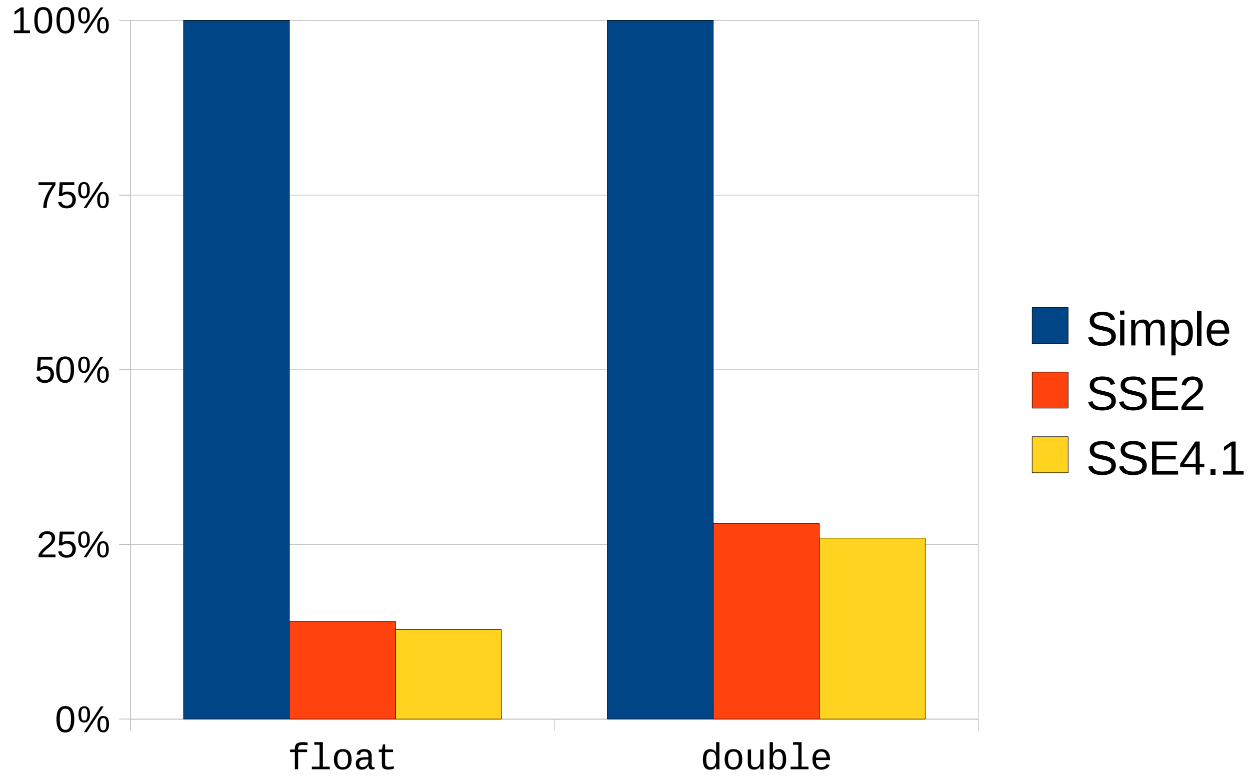
Complex Examples

- Intel adds more support:

```
void lscale(__v4sf *out, const __v4sf *in) {
  __m128 v10 = _mm_set_ps1(10.0f), v09 = _mm_set_ps1(0.9f);
  for (unsigned i = 0; i < N / 4; ++i) {
    __m128 cmp = _mm_cmp_gt(in[i], v10);
    __m128 tmp = _mm_add_ps(v10, _mm_mul_ps(_mm_sub_ps(in[i], v10),
                                            V09));
    out[i] = _mm_blendv_ps(in[i], tmp, cmp);
}
```

Benchmarking

- Performance:



- Up to 87% faster

Complex Examples

- Complex built-in operations:
  - Minimum
  - Maximum
  - Saturated arithmetic

Scalable Stream Programming

- Compilers not really optimizing automatically
- Too much lowlevel knowledge
  - Not productive enough
  - Not everybody can know the instructions
- Updating for newer CPU labor intensiv

- Better: library approach

Scalable Stream Programming

- Hide stream programming in C++ classes

```
template<typename T, int N>
struct vec {
  union {
    T n[N];
    __v4sf f[N/4]; __v2df d[N/2]; __v2di ll[N/2];
  };
  T &operator[](size_t x){return n[x];}
  T operator[](size_t x) const {return n[x];}
};
```

Scalable Stream Programming

- Hide stream programming in C++ classes

```
template<typename T, int N>
T scalar(const vec<T,N> &x, const vec<T,N> &y) {
  T r = 0;
  for (int i = 0; i < N; ++i)
    r += x[i] * y[i];
  return r;
}
```


RED HAT :: CHICAGO :: 2009 SUMMIT

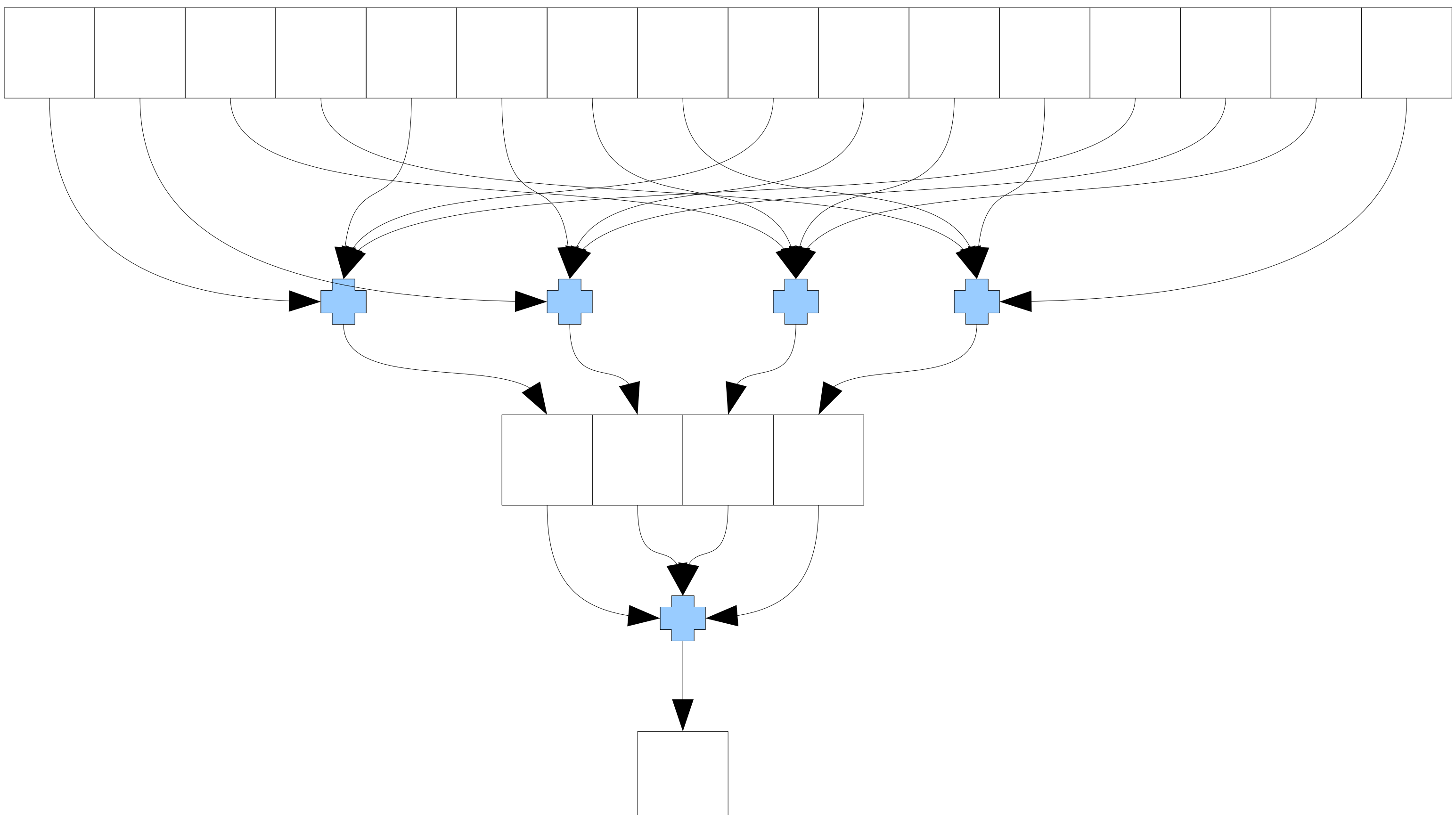
Scalable Stream Programming

- Hide stream programming in C++ classes

```
template<int N>
float scalar(const vec<float,N> &x, const vec<float,N> &y){
  __m128 t = _mm_setzero_ps();
  for (int i = 0; i < N / 4; ++i)
    t = _mm_add_ps(t, _mm_mul_ps(x.f[i], y.f[i]));
  t = _mm_hadd_ps(t, t);
  t = _mm_hadd_ps(t, t);
  return __builtin_ia32_vec_ext_v4sf(t, 0);
}
```

Scalable Stream Programming

- Hide stream programming in C++ classes



Scalable Stream Programming

- Hide stream programming in C++ classes

```
template<typename T, int N>
vec<T,N> operator+(const vec<T,N> &x, const vec<T,N> &y) {
  vec<T,N> r;
  for (int i = 0; i < N; ++i)
    r[i] = x[i] + y[i];
  return r;
}
```

Scalable Stream Programming

- Hide stream programming in C++ classes

```
template<int N>
vec<float,N> operator+(const vec<float,N> &x,
                       const vec<float,N> &y) {
  vec<float,N> r;
  for (int i = 0; i < N / 4; ++i)
    r.f[i] = _mm_add_ps(x.f[i], y.f[i]);
  return r;
}
```

Scalable Stream Programming

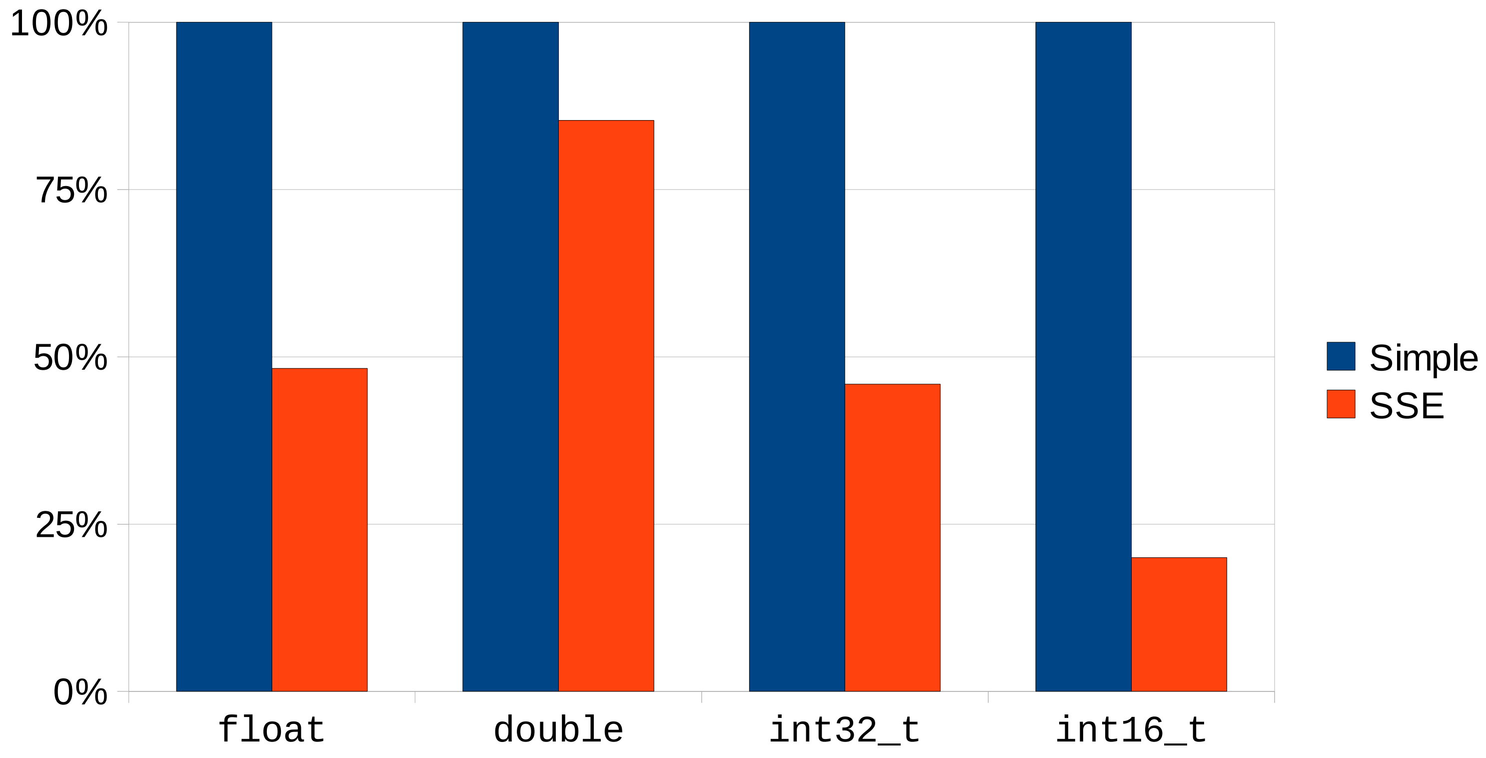
- Simple test code:

```
res = scalar(vec1 * f1 + vec2 * f2,
             vec3 * f3 + vec4 * f4);
```

- `operator*(vec, T)` defined appropriately
- Done for `float`, `double`, `int32_t`, `int16_t`

Benchmarking

- Performance of overloaded functions:



- Up to 80% faster

Scalable Stream Programming

- Avoid memory overhead:

```
template<typename T, int N>
struct factvec {
  const vec<T,N> &v;
  T f;
};
template<typename T, int N>
factvec<T,N> operator*(const vec<T,N> &v, T f)
{ return factvec(v, f); }
```

- **No code changes necessary!**

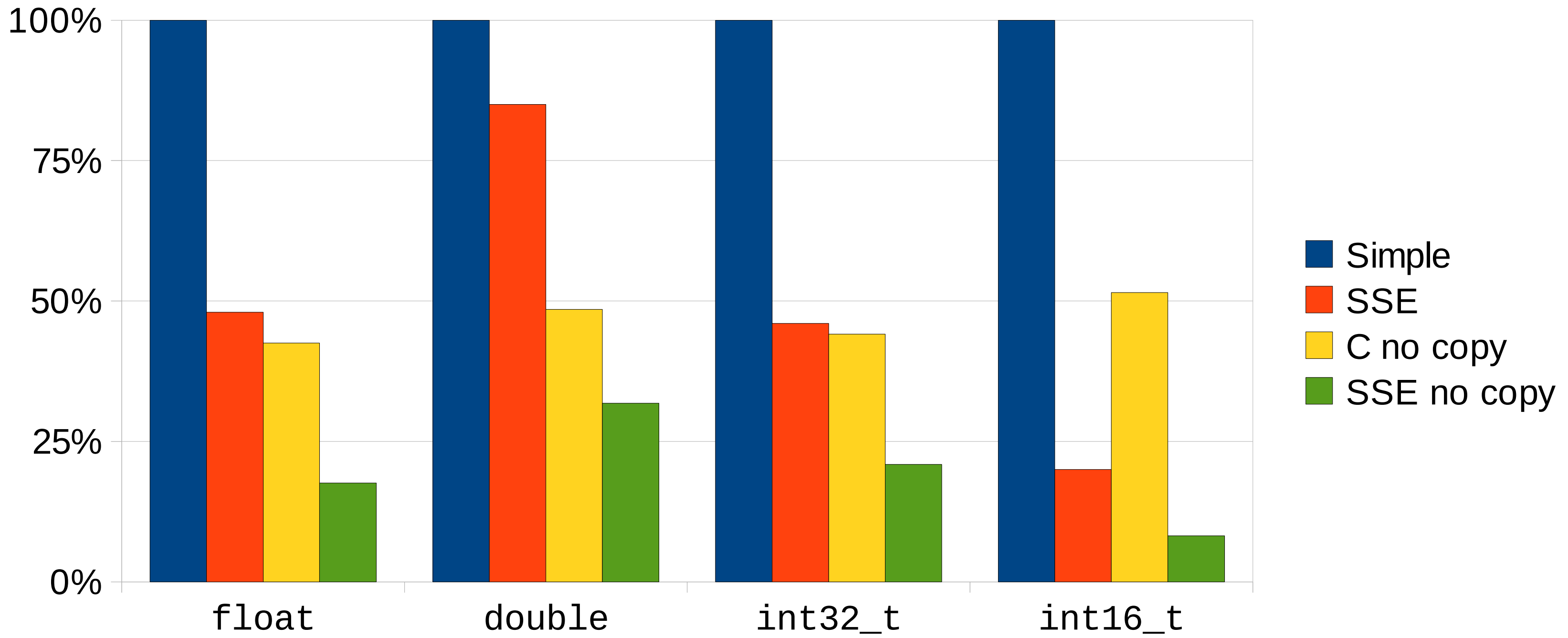
Scalable Stream Programming

- Avoid memory overhead:

```
template<int N>
vec<float,N> operator+(factvec<float,N> &x,
                       factvec<float,N> &y) {
  __m128 vxf = _mm_set_ps1(x.f), vyf = _mm_set_ps1(y.f);
  vec<float,N> r;
  for (int i = 0; i < N / 4; ++i)
    r.f[i] = _mm_add_ps(_mm_mul_ps(x.v.f[i], vxf),
                        _mm_mul_ps(y.v.f[i], vyf));
  return r;
}
```

Benchmarking

- Results when avoiding copying:



- Up to 92% faster

Scalable Stream Programming

- Delay even more:

```
template<typename T, int N>
struct sumfactvec {
  const vec<T,N> &v1; T f1;
  const vec<T,N> &v2; T f2;
};
template<typename T, int N>
sumfactvec<T,N> operator+(const factvec<T,N> &v1,
                          const factvec<T,N> &v2)
{ return sumfactvec(v1, v2); }
```

- **No code changes necessary!**

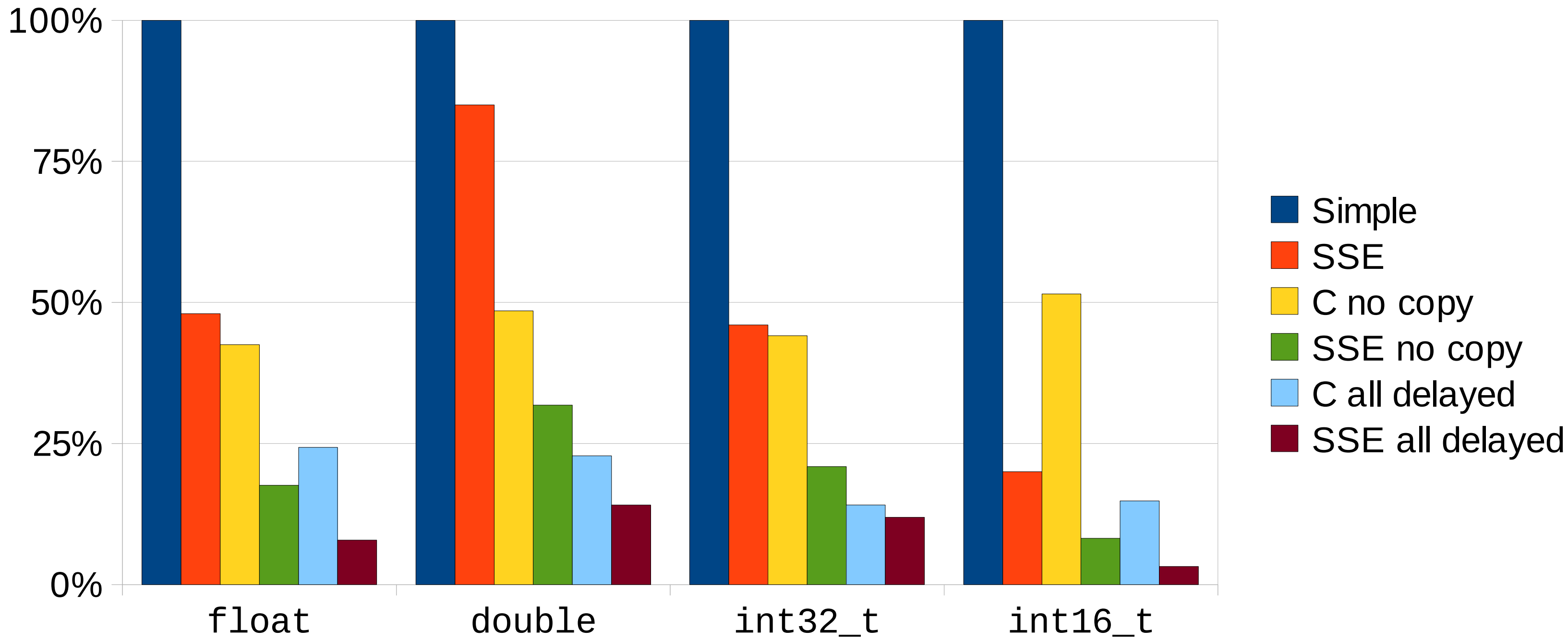
Scalable Stream Programming

- Delay even more:

```
template<int N>
float scalar(const sumfactvec<float,N> &x,
             const sumfactvec<float,N> &y)
{
  ...
}
```

Benchmarking

- Results when delaying all operations:



- Up to 97% faster

Conclusion

- Stream programming well worth it
- Normal programmers don't have to be bothered
- Specialists can modify library code during optimization
- No program changes needed after these optimizations
- C++ powerful enough to express all that's needed
  - Not showed: rvalue references (move semantics)
    - Further automatic reduction of copy operations
- gcc has full set of intrinsics to use vector instructtions
- Even wider vectors coming → more speedup

QUESTIONS?

TELL US WHAT YOU THINK:
REDHAT.COM/SUMMIT-SURVEY

drepper@redhat.com | http://people.redhat.com/drepper