tf.sparse_tensor_dense_matmul

```
sparse_tensor_dense_matmul(
    sp_a,
    b,
    adjoint_a=False,
    adjoint_b=False,
    name=None
)
```

Defined in `tensorflow/python/ops/sparse_ops.py`.

See the guide: Sparse Tensors > Math Operations

Multiply SparseTensor (of rank 2) "A" by dense matrix "B".

No validity checking is performed on the indices of `A`. However, the following input format is recommended for optimal behavior:

- If `adjoint_a == false`: `A` should be sorted in lexicographically increasing order. Use `sparse_reorder` if you're not sure.
- If `adjoint_a == true`: `A` should be sorted in order of increasing dimension 1 (i.e., "column major" order instead of "row major" order).

Using `tf.nn.embedding_lookup_sparse` for sparse multiplication:

It's not obvious but you can consider `embedding_lookup_sparse` as another sparse and dense multiplication. In some situations, you may prefer to use `embedding_lookup_sparse` even though you're not dealing with embeddings.

There are two questions to ask in the decision process: Do you need gradients computed as sparse too? Is your sparse data represented as two `SparseTensor` s: ids and values? There is more explanation about data format below. If you answer any of these questions as yes, consider using `tf.nn.embedding_lookup_sparse`.

Following explains differences between the expected SparseTensors: For example if dense form of your sparse data has shape `[3, 5]` and values:

```
[[  a      ]
 [b       c]
 [    d    ]]
```

`SparseTensor` format expected by `sparse_tensor_dense_matmul`: `sp_a` (indices, values):

```
[0, 1]: a
[1, 0]: b
[1, 4]: c
[2, 2]: d
```

`SparseTensor` format expected by `embedding_lookup_sparse`: `sp_ids` `sp_weights`

```
[0, 0]: 1                [0, 0]: a
[1, 0]: 0                [1, 0]: b
[1, 1]: 4                [1, 1]: c
[2, 0]: 2                [2, 0]: d
```

Deciding when to use `sparse_tensor_dense_matmul` vs. `matmul`(a_is_sparse=True):

There are a number of questions to ask in the decision process, including:

- Will the SparseTensor `A` fit in memory if densified?
- Is the column count of the product large (>> 1)?
- Is the density of `A` larger than approximately 15%?

If the answer to several of these questions is yes, consider converting the `SparseTensor` to a dense one and using `tf.matmul` with `a_is_sparse=True`.

This operation tends to perform well when `A` is more sparse, if the column size of the product is small (e.g. matrix-vector multiplication), if `sp_a.dense_shape` takes on large values.

Below is a rough speed comparison between `sparse_tensor_dense_matmul`, labeled 'sparse', and `matmul`(a_is_sparse=True), labeled 'dense'. For purposes of the comparison, the time spent converting from a `SparseTensor` to a dense `Tensor` is not included, so it is overly conservative with respect to the time ratio.

Benchmark system: CPU: Intel Ivybridge with HyperThreading (6 cores) dL1:32KB dL2:256KB dL3:12MB GPU: NVidia Tesla k40c

Compiled with: `-c opt --config=cuda --copt=-mavx`

```
tensorflow/python/sparse_tensor_dense_matmul_op_test --benchmarks
A sparse [m, k] with % nonzero values between 1% and 80%
B dense [k, n]

% nnz  n   gpu   m     k     dt(dense)     dt(sparse)   dt(sparse)/dt(dense)
0.01   1   True  100   100   0.000221166   0.00010154   0.459112
0.01   1   True  100   1000  0.00033858    0.000109275  0.322745
0.01   1   True  1000  100   0.000310557   9.85661e-05  0.317385
0.01   1   True  1000  1000  0.0008721     0.000100875  0.115669
0.01   1   False 100   100   0.000208085   0.000107603  0.51711
0.01   1   False 100   1000  0.000327112   9.51118e-05  0.290762
0.01   1   False 1000  100   0.000308222   0.00010345   0.335635
0.01   1   False 1000  1000  0.000865721   0.000101397  0.117124
0.01   10  True  100   100   0.000218522   0.000105537  0.482958
0.01   10  True  100   1000  0.000340882   0.000111641  0.327506
0.01   10  True  1000  100   0.000315472   0.000117376  0.372064
0.01   10  True  1000  1000  0.000905493   0.000123263  0.136128
0.01   10  False 100   100   0.000221529   9.82571e-05  0.44354
0.01   10  False 100   1000  0.000330552   0.000112615  0.340687
0.01   10  False 1000  100   0.000341277   0.000114097  0.334324
0.01   10  False 1000  1000  0.000819944   0.000120982  0.147549
0.01   25  True  100   100   0.000207806   0.000105977  0.509981
0.01   25  True  100   1000  0.000322879   0.00012921   0.400181
0.01   25  True  1000  100   0.00038262    0.00014158   0.370035
0.01   25  True  1000  1000  0.000865438   0.000202083  0.233504
0.01   25  False 100   100   0.000209401   0.000104696  0.499979
0.01   25  False 100   1000  0.000321161   0.000130737  0.407076
0.01   25  False 1000  100   0.000377012   0.000136801  0.362856
0.01   25  False 1000  1000  0.000861125   0.00020272   0.235413
0.2    1   True  100   100   0.000206952   9.69219e-05  0.46833
0.2    1   True  100   1000  0.000348674   0.000147475  0.422959
0.2    1   True  1000  100   0.000336908   0.00010122   0.300439
0.2    1   True  1000  1000  0.001022      0.000203274  0.198898
0.2    1   False 100   100   0.000207532   9.5412e-05   0.459746
0.2    1   False 100   1000  0.000356127   0.000146824  0.41228
0.2    1   False 1000  100   0.000322664   0.000100918  0.312764
0.2    1   False 1000  1000  0.000998987   0.000203442  0.203648
0.2    10  True  100   100   0.000211692   0.000109903  0.519165
0.2    10  True  100   1000  0.000372819   0.000164321  0.440753
0.2    10  True  1000  100   0.000338651   0.000144806  0.427596
```

```
0.2    10  True  1000  1000  0.00108312    0.000758876  0.70064
0.2    10  False 100   100   0.000215727   0.000110502  0.512231
0.2    10  False 100   1000  0.000375419   0.0001613    0.429653
0.2    10  False 1000  100   0.000336999   0.000145628  0.432132
0.2    10  False 1000  1000  0.00110502    0.000762043  0.689618
0.2    25  True  100   100   0.000218705   0.000129913  0.594009
0.2    25  True  100   1000  0.000394794   0.00029428   0.745402
0.2    25  True  1000  100   0.000404483   0.0002693    0.665788
0.2    25  True  1000  1000  0.0012002     0.00194494   1.62052
0.2    25  False 100   100   0.000221494   0.0001306    0.589632
0.2    25  False 100   1000  0.000396436   0.000297204  0.74969
0.2    25  False 1000  100   0.000409346   0.000270068  0.659754
0.2    25  False 1000  1000  0.00121051    0.00193737   1.60046
0.5    1   True  100   100   0.000214981   9.82111e-05  0.456836
0.5    1   True  100   1000  0.000415328   0.000223073  0.537101
0.5    1   True  1000  100   0.000358324   0.00011269   0.314492
0.5    1   True  1000  1000  0.00137612    0.000437401  0.317851
0.5    1   False 100   100   0.000224196   0.000101423  0.452386
0.5    1   False 100   1000  0.000400987   0.000223286  0.556841
0.5    1   False 1000  100   0.000368825   0.00011224   0.304318
0.5    1   False 1000  1000  0.00136036    0.000429369  0.31563
0.5    10  True  100   100   0.000222125   0.000112308  0.505608
0.5    10  True  100   1000  0.000461088   0.00032357   0.701753
0.5    10  True  1000  100   0.000394624   0.000225497  0.571422
0.5    10  True  1000  1000  0.00158027    0.00190898   1.20801
0.5    10  False 100   100   0.000232083   0.000114978  0.495418
0.5    10  False 100   1000  0.000454574   0.000324632  0.714146
0.5    10  False 1000  100   0.000379097   0.000227768  0.600817
0.5    10  False 1000  1000  0.00160292    0.00190168   1.18638
0.5    25  True  100   100   0.00023429    0.000151703  0.647501
0.5    25  True  100   1000  0.000497462   0.000598873  1.20386
0.5    25  True  1000  100   0.000460778   0.000557038  1.20891
0.5    25  True  1000  1000  0.00170036    0.00467336   2.74845
0.5    25  False 100   100   0.000228981   0.000155334  0.678371
0.5    25  False 100   1000  0.000496139   0.000620789  1.25124
0.5    25  False 1000  100   0.00045473    0.000551528  1.21287
0.5    25  False 1000  1000  0.00171793    0.00467152   2.71927
0.8    1   True  100   100   0.000222037   0.000105301  0.47425
0.8    1   True  100   1000  0.000410804   0.000329327  0.801664
0.8    1   True  1000  100   0.000349735   0.000131225  0.375212
0.8    1   True  1000  1000  0.00139219    0.000677065  0.48633
0.8    1   False 100   100   0.000214079   0.000107486  0.502085
0.8    1   False 100   1000  0.000413746   0.000323244  0.781261
0.8    1   False 1000  100   0.000348983   0.000131983  0.378193
0.8    1   False 1000  1000  0.00136296    0.000685325  0.50282
0.8    10  True  100   100   0.000229159   0.00011825   0.516017
0.8    10  True  100   1000  0.000498845   0.000532618  1.0677
0.8    10  True  1000  100   0.000383126   0.00029935   0.781336
0.8    10  True  1000  1000  0.00162866    0.00307312   1.88689
0.8    10  False 100   100   0.000230783   0.000124958  0.541452
0.8    10  False 100   1000  0.000493393   0.000550654  1.11606
0.8    10  False 1000  100   0.000377167   0.000298581  0.791642
0.8    10  False 1000  1000  0.00165795    0.00305103   1.84024
0.8    25  True  100   100   0.000233496   0.000175241  0.75051
0.8    25  True  100   1000  0.00055654    0.00102658   1.84458
0.8    25  True  1000  100   0.000463814   0.000783267  1.68875
0.8    25  True  1000  1000  0.00186905    0.00755344   4.04132
0.8    25  False 100   100   0.000240243   0.000175047  0.728625
0.8    25  False 100   1000  0.000578102   0.00104499   1.80763
0.8    25  False 1000  100   0.000485113   0.000776849  1.60138
0.8    25  False 1000  1000  0.00211448    0.00752736   3.55992
```

Args:

- `sp_a` : SparseTensor A, of rank 2.
- `b` : A dense Matrix with the same dtype as sp_a.
- `adjoint_a` : Use the adjoint of A in the matrix multiply. If A is complex, this is transpose(conj(A)). Otherwise it's transpose(A).
- `adjoint_b` : Use the adjoint of B in the matrix multiply. If B is complex, this is transpose(conj(B)). Otherwise it's transpose(B).
- `name` : A name prefix for the returned tensors (optional)

Returns:

A dense matrix (pseudo-code in dense np.matrix notation): **`A = A.H if adjoint_a else A`** **`B = B.H if adjoint_b else B`** **`return A*B`**
